版权注意事项：
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

HZ BOOKS
华章IT

国内第一本全面讲解Web前端自动化构建的实用指南，涵盖前端开发所需的各种工具

全方位梳理前端构建的整体概念和思路，深入探索使用Gulp、Bower和Yeoman等工具自动高效开发完整工作流的各种方法和技巧

MANNING

Web开发技术丛书

# Front-End Tooling

with Gulp, Bower and Yeoman

[奥] 斯特凡·鲍姆加特纳 (Stefan Baumgartner) 著
谈博文 译

# Web前端自动化构建

## Gulp、Bower和Yeoman开发指南

机械工业出版社
China Machine Press

本书重点关注的是广义的开发工作流，涵盖了开发者开发和维护项目时将会面临的三个阶段，分别是创建、开发和部署。对于每个阶段，本书都详细解释了这一阶段的需求和概念，并且介绍了一个合适的工具来实现这些需求。

全书共10章。第1章展现了前端开发者日常面临的挑战，并且介绍了如何用三种类型的工具建立工作流以克服这些挑战。第2章介绍了第一种工具，也就是构建工具Gulp。第3章介绍依赖和执行链。第4章引入依赖管理的概念，并选择Bower作为依赖管理的工具。第5章是前端工作流的最后一步，介绍如何通过脚手架工具 Yeoman来复用第2章到第4章中搭建好的流程。第6章介绍如何利用Gulp的特性来改变Gulp在不同开发环境中的输出结果。第7章深入解释 Gulp 的底层技术——文件对象流。第8章介绍了Gulp 的插件生态系统，并且告诉读者在选择新插件作为构建管道时，如何避免冗余的或者质量不好的插件。第9章介绍了模块的概念，并且展示几个不同的模块规范之间的差异。第10章以脚手架工具Yeoman来结束全书。

# Web前端自动化构建

## Gulp、Bower和Yeoman开发指南

[奥] 斯特凡·鲍姆加特纳（Stefan Baumgartner）著
谈博文 译

# Front-End Tooling

with Gulp, Bower and Yeoman

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Web 前端自动化构建：Gulp、Bower 和 Yeoman 开发指南 /（奥）斯特凡·鲍姆加特纳（Stefan Baumgartner）著；谈博文译 . —北京：机械工业出版社，2017.9
（Web 开发技术丛书）
书名原文：Front-End Tooling with Gulp, Bower and Yeoman

ISBN 978-7-111-57883-3

I. W… II. ①斯… ②谈… III. 超文本标记语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2017）第 209489 号

本书版权登记号：图字：01-2017-0731

Stefan Baumgartner: *Front-End Tooling with Gulp, Bower and Yeoman* (ISBN 978-1617292743).

Original English edition published by Manning Publications Co., 209 Bruce Park Avenue, Greenwich, Connecticut 06830.

Copyright © 2017 by Manning Publications Co.

All rights reserved.

Simplified Chinese translation edition published by China Machine Press.

Copyright © 2017 by China Machine Press.

本书中文简体字版由 Manning 出版公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

# Web 前端自动化构建：Gulp、Bower 和 Yeoman 开发指南

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：和　静　　责任校对：李秋荣
印　　刷：三河市宏图印务有限公司　　版　　次：2017 年 9 月第 1 版第 1 次印刷
开　　本：186mm×240mm　1/16　　印　　张：13.75
书　　号：ISBN 978-7-111-57883-3　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

版权所有 • 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

# *The Translator's Words* 译者序

遥想当年用 jQuery 刀耕火种的时代，如今的前端开发者幸福太多了。从 UI 组件到前端框架，从语法糖到预处理语言，前端生态圈的变化可谓是日新月异。但是别看现在的开发者熟稔 React、Vue、ES6/ES7 等技术，假如让他从零开始开发一个项目，他可能就无从下手、不知所措了。

原因就在于，现在的前端项目的构建太复杂了。先不论前端框架，ES6 这么流行，不上个 babel 简直说不过去。还有 CSS，暂且不说 PostCSS，Sass 总得来一个吧。等到要上生产环境，合并、混淆也是必不可少的……搭建一个现代 Web 应用所需要的工具太多了，实在列举不完。这其中步骤之复杂、细节之烦琐，每一个搭建过前端构建流程的朋友肯定都深有体会。

对于初学者而言，前端工具更是难以入门。绝大部分开发者一般都没有机会参与搭建构建流程的环节。而且一般只有在项目初始阶段才需要搭建流程，之后就不会再去修改它。等到开始下一个项目时，构建工具又已经发生了翻天覆地的变化。这就导致了前端构建方面的知识难以积累和沉淀。

本书就非常适合前端构建的初学者。本书所介绍的 Gulp、Bower 和 Yeoman 都是业内流行且易于上手的工具。而且本书的目的并不仅仅是介绍工具的用法，还对前端构建整体的概念和思路进行了梳理。即使未来出现了更先进、更时髦的工具，总体的思路依然万变不离其宗，本书也不会因此而过时。

译者在翻译的时候力求用词准确、行文流畅，但也难免有所纰漏，恳请广大读者来信批评指正（tanbowensg@gmail.com）。

# 自 序 *Foreword*

我十几岁的时候就喜欢上网。正如所有小时候的爱好一样，网络对我也产生了深远的影响。在网上人们几乎可以发布任何信息，传播开放的思想。无论是谁在网络上都有发声的权利（无论是好是坏）。正是这一点吸引了我，我也因此从网络的用户成为了一名开发者。

我曾经深入钻研过《勇者斗恶龙》这款游戏。我曾在 11 年间，在网上发布了超过 1500 页的游戏攻略。一种对于文字和电影的热情驱使着我制作 podcast 来表达自己的想法，这一过程又持续了 4 年。在我硬盘的某个角落里，至今还存有以前未完成的视频原型，幸好它们永远也不会有机会重见天日了。我所有的爱好和疯狂的想法都以某种形式发布到了网上，而且也一直都受到好评。

直到 2010 年（可以说是任天堂的一年），我才意识到我最大的爱好并不是上面提到的那些，尽管我为它们也付出了很多。曾经有人邀请我给更多电脑游戏写攻略，当时我有点膨胀、骄傲、意气风发，但是很快我就意识到，我已经不像以前那样对电脑游戏那么感兴趣了，我再也不可能写出以前那么高质量的攻略了。但是我可以把精力用在创作所有作品所使用的工具上，让别人来创作出更好的作品。那就是所有网站的基础——HTML、CSS 和 JavaScript。

我辞去了图像处理的工作，我一直对它不感兴趣。从此以后我都管自己叫网站开发者。我很幸运地被当地的一家网络公司雇用了，能够从事自己喜欢的工作。那是我有生之年最正确的决定。不久之后，我还有幸领导过几个世界上最棒的前端开发团队。我发现在工作中最有挑战性的事情是：制定代码规范、项目架构和自动化流程，并且以一种高效而温和的方式推广给团队成员。这些技术并不是鸡毛蒜皮的事情，它们能

真正改善开发体验。我们尝试了很多工具，但是真正取得进展还是在 Yeoman 问世之后。从此以后，对于每一个新项目我都会用到 JavaScript 构建工具、依赖管理工具和脚手架工具，来保证项目能顺利地开始。

就像这本书中，展现了我和网络的缘份。我是由于喜欢发表东西才喜欢上网络，最后又回到了本源，发表了关于网络技术本身的东西。而且这次不是在网上发表，而是印刷出版了，这在几年前是我完全无法想象的。

如今网络还在影响着我。通过网络，我认识了遍布世界各地的朋友。网络还让我有机会在各个国家和城市之间旅行。甚至我还通过网络认识了我的妻子，未来网络一定还能给我更多惊喜。

# 前　言 *Preface*

本书重点关注的是广义的开发工作流，涵盖了开发者开发和维护项目时将会面临的三个阶段，分别是初始化、开发和部署。对于每个阶段，本书都详细解释了这一阶段的需求和概念，并且介绍了一个合适的工具来实现这些需求。

在现代 Web 开发中，新的工具层出不穷，但是寿命往往都很短。我写本书花了超过一年的时间，这在 JavaScript 界可能相当于过了十年。工具日新月异的变化对写书提出了巨大的挑战，但同时也提醒了我不要过于关注工具本身。因此，本书介绍的每个技术都可以轻松地替换为另一种合适的工具，而不会影响关键的概念。另一方面，本书挑选工具的标准不是工具的流行程度，而是工具的可维护性（sustainability）。本书介绍的工具都被广泛使用，且具有一定的用户基础和宽广的应用空间。正是因为这些特点，本书在几年后依然会有参考价值。

## 目标读者

本书的目标读者是想要在日常工作流中添加自动化流程和工具链的开发者。本书建立的工作流模板既适合初学者，也适合有经验的开发者。尽管市面上已经有了许多教程和博文来介绍这些工具，但大都只是单独介绍其中的某一个。而本书则把所有这些工具融合到一个工作流中，绝不仅仅只是介绍工具用法这些表面知识而已。

## 本书大纲

本书分为两部分，每一部分包括五章。

第一部分介绍一套全新前端开发工作流和相关的工具，不仅描述了如何用三个专门的工具来搭建一个工作流，还介绍了前端开发工作流的一些总体构想。建议按照顺

序来读这一部分。

- 第 1 章展现了前端开发者日常面临的挑战，并且介绍了如何用三种类型的工具建立工作流以克服这些挑战。
- 第 2 章介绍了第一种工具，也就是构建工具。它提供了抽象接口来自动化处理底层代码的任务（task）。这一章的内容主要基于 Gulp。
- 第 3 章介绍依赖（dependency）和执行链（execution chain）。这一章会结合上一章的任务，把它们添加到一系列的执行中，以此来建立一个本地的开发环境。
- 第 4 章引入依赖管理的概念。这一章会介绍对于前端开发而言，扁平依赖树和嵌套依赖树之间的区别和优劣。这一章选择 Bower 作为依赖管理的工具。
- 第 5 章是前端工作流的最后一步。这一章会介绍如何通过脚手架工具 Yeoman，来复用第 2 章到第 4 章中搭建好的流程。

读完第一部分，读者就能够运用上述的三种工具来改进工作流，以适应自己的需求了。所有介绍的工具都使用 Node.js 作为运行时。刚刚接触 Node.js 的开发者可以参考附录来大致了解一下。

第一部分中介绍的 Gulp、Bower 和 Yeoman 都可以轻松地替换为其他工具。不过，第二部分就会深入探讨这三个工具的内在工作原理。其中每一章都可以提高工作流的效率，还会说明这些工具的独特和有用之处。

- 第 6 章会介绍如何利用 Gulp 的特性，包括增量编译（incremental build）和管道开关（pipeline switch），来改变 Gulp 在不同开发环境中的输出结果。
- 第 7 章深入解释了 Gulp 的底层技术——文件对象流（file object stream）。使用这一技术可以在不同的输入 / 输出场景中复用相似的管道。
- 第 8 章介绍了 Gulp 的插件生态系统，并且告诉读者在选择新插件作为构建管道（build pineline）时，如何避免冗余的或者质量不好的插件。此外还介绍了如何通过 Gulp 的 task 和 streaming API 来集成插件以外的工具。
- 第 9 章又回过来讲依赖管理。这一章介绍了模块（module）的概念，并且展示几个不同的模块规范之间的差异，模块能让我们更轻松地把依赖集成到应用中。
- 第 10 章以脚手架工具（scaffolding tool）Yeoman 来结束全书。在 Yeoman sub-generator 的帮助下，项目的脚手架不仅在项目初期十分有用，并且会贯穿整个项目的始终。

第二部分的目的是让读者精通这些工具。尽管这些工具的示例都是有联系的，但

读者也可以跳着阅读分别来了解它们的概念和细节。

## 代码格式和下载

本书包含了很多示例的源代码，有时会加粗代码，来凸显和前几章代码的不同之处，比如向代码中加入新功能。

大多数情况下，原始的源代码都被格式化过。本书在代码中加入了换行和缩进来适应本书的版面。在极端情况下，偶尔还会使用续行符（line-continuation marker）。另外，当源代码在正文中出现时，会移除源代码中的注释。包含重要概念的代码会附上注释。

本书中示例的源代码可以在出版社的网站 www.manning.com/books/front-end-tooling-with-gulp-bower-and- yeoman 上下载。

本书中的示例也可以从 GitHub 网站 https://github.com/frontend-tooling 上下载。项目“sample-project-gulp”中的分支中包含了每一章的解决方案。

## 网上资源

- StackOverflow 上的 Gulp.js 标签（http://stackoverflow.com/questions/tagged/gulp）里有很多问题和回答。上面的网友都非常友好而且见多识广，能够很好地回答一些特殊的问题。
- Yeoman 网站（http://yeoman.io）不仅包含 Yeoman 自己的资源，也提供了关于其他工具的丰富的教程资源。
- 作者的博客上也有很多 Gulp 和 Yeoman 的教程：https://fettblog.eu。这些文章更加简明扼要，解决的都是更为具体的问题。

# *Acknowledgements* 致谢

你可能觉得写书是一项孤独的工作，但事实却并非如此。

在所有帮助过我的人中，首先要提到的就是 Lesley Trites。作为我在 Manning 的策划编辑，她给了我非常有价值的反馈，是我写作时最棒的伙伴。她手把手地教我写作的技巧，如果没有她，本书根本不会像现在这个样子。

说到 Manning 的编辑们，我跟技术编辑 Johan Pretorious 和 Nick Watts 也合作得非常愉快！感谢你们检查我提出的想法。

还要感谢那些在我的写作过程中提出了很多有用反馈的读者们，他们是：Andy Knight、David DiMaria、Giancarlo Massari、Harinath Mallepally、Jason Gretz、Jeroen Benckhuijsen、Johan Pretorius、Mario Ruiz、Nikander 和 Margriet Bruggeman、Palak Mathur、Tanya Wilke、Unnikrishnan Kumar 和 Zorodzayi Mukuya。

Alexander Zaytsev、Lars Johansen 和 Jens Klinger，他们怀着对开源的高涨热情，在我写书的时候，一直都在改进我在 GitHub 上的 Gulp 示例。

我所有关于前端工作流的知识都是 Addy Osmani 教我的。我从 Sebastian Gierlinger 和 Blaine Bublitz 那里知道了什么是开源。

Thomas Pink 和 Thomas Heller 不仅是我最棒的同事，而且还一直提出有价值的问题，让我改进构建工具技术。他们对技术改进的推动和现实工作的具体需求是本书内容来源的坚实基础。

还要感谢“Working Draft”podcast 上的热心朋友，他们是 Christian Schaefer、Peter Kröner、Rodney Rehm、Anselm Hannemann 和 Hans-Christian Reinl，他们欢乐和富有挑战性的讨论让我不放过文中的每一个细节。同时还要感谢“Klassenfahrt”团队：Fabian

Beiner、Sven Wolfermann、Sebastian Golasch、Der Pepo、Robert Weber、Marc Hinse、Bianca Kastl、Joschi Kuphal、Marc Thiele、Khalil Lechelt、Frederic Hemberger、Tobias Baldauf、Tom Arnold 和 Maik Wagner。没有你们我绝不会取得今天的成就。

人们总是对作家有一种浪漫的幻想，觉得他们每周都在法国普罗旺斯的小木屋中写作。为了满足读者的幻想，我要感谢我的岳父岳母 Hans 和 Marianne，因为每隔一周的周末我都是在他们奥地利 Innviertel 的家中度过的。我还要感谢我的父母 Hans 和 Rosi 平时对我的照顾！感谢你们的支持！

最后，我最想感谢的人是我最爱的妻子 Doris，她的耐心、爱意和关心对我来说无可替代。Doris，我欠你太多了。

## *about the cover illustration* 关于原书封面插图

本书封面插图的标题为“穿着中世纪衣服的男人”。这幅插图的作者是 Paolo Mercuri（1804—1884），选自“Costumes Historiques”。这是一本多卷历史服饰简史，时间横跨 12 世纪到 15 世纪，由 Camille Bonnard 整理和编辑，并于 19 世纪 50 年代或 60 年代在巴黎出版。19 世纪的人们对异国情调的兴趣日益增加，人们沉迷于这类收藏，因为这可以帮助他们探索所生活的世界和很久以前的世界。

Mercuri 的这部历史藏书中丰富多彩的插图生动地展示了几百年前世界各地的城市和地区的文化差异。在街头或农村，仅仅通过服装，就可以识别出人们的生活状态、贸易和职业。几个世纪以来，着装习俗发生了变化。曾经的各个地区的服装的多样性，现在都已经渐渐消失了。如今我们已经很难区别来自不同大陆的居民，更不要说区别他们来自的城市和国家，以及社会地位和职业了。也许如今文化多样性已经被更为多样的个人生活所取代——一种更加多样、节奏更快的科技生活。

在这样一个计算机书籍逐渐趋同的时代，Manning 利用几个世纪之前不同地域丰富多样的生活来制作图书封面，借此赞美计算机行业的创造力和引领时代的精神。

# 目　录 *Contents*


译者序
自　序
前　言
致　谢
关于原书封面插图

## 第一部分　现代 Web 应用工作流

### 第 1 章　现代前端工作流的工具链 …… 3

1.1　软件开发者的工作流和任务清单 …… 4
1.1.1　初始化阶段的任务 …… 6
1.1.2　开发阶段的任务 …… 6
1.1.3　部署阶段的任务 …… 7
1.1.4　人为因素 …… 8
1.2　JavaScript 工具和新的工作流 …… 9
1.2.1　三类代码及其工具 …… 10
1.2.2　Node.js 中的 JavaScript 工具 …… 12
1.3　用 Yeoman 搭建脚手架 …… 13
1.3.1　Yeoman 作为脚手架工具的优点 …… 14
1.3.2　什么是 generator …… 14
1.4　依赖管理工具 Bower …… 15
1.4.1　Bower 的优点 …… 16
1.4.2　Bower 的依赖树 …… 17
1.5　Gulp 流式构建系统 …… 18
1.5.1　Gulp 的优点 …… 18
1.5.2　构建管道 …… 19
1.6　总结 …… 21

### 第 2 章　Gulp 入门 …… 22

2.1　配置 Gulp …… 23
2.1.1　Gulp 的构建块 …… 23
2.1.2　Gulp 命令行接口 …… 25
2.1.3　安装本地 Gulp …… 26
2.2　创建 Gulpfile …… 28
2.2.1　Gulp 的“Hello World” task …… 28

2.2.2 流的使用 …… 31
2.2.3 Gulp 中的可读流和可写流 …… 32
2.3 使用 Gulp 插件处理 task …… 34
2.3.1 转换数据 …… 34
2.3.2 改变文件结构 …… 35
2.3.3 链式调用插件 …… 37
2.4 总结 …… 39

**第 3 章 用 Gulp 配置本地开发环境 …… 41**

3.1 本地开发环境 …… 42
3.1.1 给 task 分类 …… 43
3.1.2 重置步骤：clean task …… 43
3.1.3 构建和迭代 …… 45
3.2 初次构建步骤 …… 45
3.2.1 依赖链和执行链 …… 46
3.2.2 顺序和并行执行 task …… 47
3.2.3 依赖链的 task 执行函数 …… 48
3.2.4 更新 Gulpfile …… 49
3.3 迭代步骤 …… 52
3.3.1 监控过程 …… 52
3.3.2 使用 Browsersync 建立按需实时重载的服务器 …… 53
3.3.3 更新 Gulpfile …… 55
3.4 总结 …… 57

**第 4 章 Bower 依赖管理 …… 59**

4.1 依赖管理基础 …… 61
4.1.1 组件和依赖 …… 61
4.1.2 依赖树 …… 62
4.2 Bower 的扁平依赖树管理 …… 64
4.2.1 安装和更新组件 …… 65
4.2.2 安装有依赖的组件 …… 66
4.2.3 解决依赖冲突 …… 68
4.3 集成 Gulp …… 69
4.3.1 用 Gulp 把依赖集成到应用中 …… 70
4.3.2 将依赖集成到 Gulp 构建流程中 …… 72
4.4 总结 …… 74

**第 5 章 脚手架工具 Yeoman …… 75**

5.1 Yeoman generator …… 76
5.1.1 generator 的构成 …… 78
5.1.2 调用 generator …… 79
5.2 项目模板 …… 81
5.2.1 固定文件 …… 81
5.2.2 灵活文件 …… 83
5.2.3 可选文件 …… 83
5.2.4 可恢复文件 …… 84
5.3 Yeoman 的组装流程 …… 84
5.3.1 安装提示 …… 85
5.3.2 写入文件 …… 87
5.3.3 安装依赖 …… 89
5.4 创建一个 generator …… 89
5.4.1 准备项目模板 …… 90
5.4.2 创建组装指令 …… 92
5.4.3 构建和测试 …… 93
5.5 总结 …… 94

## 第二部分　工具链的集成和扩展


**第 6 章　不同环境下的 Gulp** …… 97

6.1　增量构建和构建缓存 …… 99

6.1.1　Gulp 内置功能 …… 99

6.1.2　安装构建缓存 …… 101

6.1.3　处理被删除的文件 …… 104

6.2　通过 sourcemap 来调试源文件 …… 105

6.2.1　什么是 sourcemap …… 107

6.2.2　Gulp 内置的 sourcemap …… 108

6.2.3　sourcemap 插件 …… 109

6.3　切换环境 …… 111

6.3.1　noop() …… 111

6.3.2　dev() 和 prod() 选择函数 … 113

6.3.3　传递命令行参数 …… 115

6.4　总结 …… 116

**第 7 章　使用流** …… 117

7.1　处理不同类型的输入 …… 119

7.1.1　穿插流 …… 119

7.1.2　合并流 …… 121

7.2　处理不同的输出 …… 123

7.2.1　在 task 中根据参数创建流 …… 123

7.2.2　流数组 …… 126

7.3　其他关于流的使用技巧 …… 128

7.3.1　使用 snippet 来避免重复代码 …… 129

7.3.2　使用流队列来管理顺序 … 130

7.3.3　通过 Gulp filter 来修改流的内容 …… 131

7.4　总结 …… 133

**第 8 章　扩展 Gulp** …… 134

8.1　Gulp 插件黑名单 …… 135

8.1.1　理由：不是一个 Gulp 插件 …… 136

8.1.2　理由：可以用别的东西替代 …… 137

8.1.3　理由：做的事情太多 …… 138

8.2　集成其他基于流的工具 …… 139

8.2.1　流、缓冲和 Vinyl 文件对象 …… 139

8.2.2　集成 Browserify …… 142

8.2.3　转换内容 …… 145

8.3　集成基于 Promise 的工具 …… 146

8.3.1　Promise 的用法 …… 147

8.3.2　在 Gulp task 系统中使用 Promise …… 149

8.4　总结 …… 151

**第 9 章　创建模块和 Bower 组件** 153

9.1　异步模块规范 …… 155

9.1.1　AMD——为浏览器设计的模块定义规范 …… 155

9.1.2　重构到 AMD …… 158

9.2　CommonJS 和通用模块规范 …… 160

9.2.1　CommonJS 模块 …… 160

9.2.2 通用模块规范 …… 161
9.3 定义和部署 Bower 组件 …… 163
9.3.1 Bower.json 说明 …… 164
9.3.2 部署和发布组件 …… 166
9.3.3 私有项目仓库和组件 …… 167
9.4 总结 …… 169

**第 10 章 Yeoman generator 进阶** …… 171

10.1 向现有项目中添加新文件 …… 172
10.1.1 sub-generator 的概念 …… 172
10.1.2 sub-generator 的组装指令 …… 174
10.2 改进接口 …… 176
10.2.1 修改现有的文件 …… 177
10.2.2 处理用户的配置 …… 178
10.3 组合 generator …… 181
10.3.1 全局的 generator 的组合 …… 181
10.3.2 依赖和 generator 的组合 …… 184
10.4 总结 …… 186

**附录 A Node.js 导论** …… 187

第一部分 Part 1

# 现代 Web 应用工作流


- 第 1 章　现代前端工作流的工具链
- 第 2 章　Gulp 入门
- 第 3 章　用 Gulp 配置本地开发环境
- 第 4 章　Bower 依赖管理
- 第 5 章　脚手架工具 Yeoman

第一部分介绍的是前端开发的工作流和工具链。这一部分中，读者将会了解前端开发者在日常工作中所要面对的挑战，并且将会学习如何通过三个工具实现一个工作流来克服这些挑战。

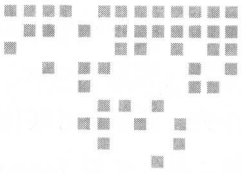
第1章介绍一套本书所推荐的工作流，以及用于管理这套工作流的工具。

第2章介绍第一个工具：构建工具Gulp。借助Gulp，你可以搭建一个测试和编译JavaScript和CSS文件的自动化流程。

第3章在第2章的基础上，把第2章的任务组织成执行链的形式。除此以外，还会搭建一个实时加载的开发服务器，它能够把构建的文件更新到本地开发环境中。

第4章介绍第二个工具：依赖管理工具Bower。在这一章中，读者将学习组件和依赖树方面的知识，以及如何使用Bower来管理第三方软件。

第5章会结合前几章的任务，通过脚手架工具Yeoman把这些任务进行打包。借助Yeoman，你可以打包出能够立刻开始开发的项目模板，以备将来使用。



第 1 章 Chapter 1

# 现代前端工作流的工具链

**本章包括：**

- 前端开发所面对的单调重复的工作。
- 脚手架和模板代码。
- 依赖管理工具。
- 构建工具。
- 前端开发的工具链和工作流。

前端开发者们赶上了一个好时代。如今，浏览器的功能比以往更强大。新技术 HTML5 的到来给前端带来了无限可能。JavaScript 再也不是一个玩具语言了，它能够开发各种各样的应用。越来越多的人采用胖客户端[⊖]（fat client）的应用架构，并认为它是一个优秀的开发 Web 应用的架构。

随着越来越多人重视 JavaScript，为了给应用写出更好的代码，开发者也面临着不少挑战：运行代码编译工具（authoring tool）、打包文件、压缩应用大小、优化图片……开发者需要合适的工具，才能满足这些需求。

这一章会考察一些编程以外、但在日常工作流中开发者又无法避免的任务，并且说明工具对处理这些任务有何帮助。我们把这些任务分成三类，然后用三种工具来解决它们：

⊖ 胖客户端是在浏览器中直接运行应用的主要代码的应用架构。

- 构建系统，比如 Gulp。只需要按一下按钮，它就能处理大量文件的更改。
- 依赖管理，比如 Bower。它能管理常用的类库的版本信息，提醒开发者类库之间的冲突，甚至有时候自己就能把冲突解决了。
- 脚手架工具，比如 Yeoman。它能提供一个项目运行所需的基本文件。只需要在控制台输入一行命令，就能用 Yeoman 创建一个全新的应用或模块。

这三个工具代表了前端工具链的基本理念：通过一系列必需的工具，来启动、运行和部署应用。在本书中，读者不仅会学到如何为自己和同事来配置这些工具，而且还会深入到每个工具中去，从而学会如何结合这些工具量身打造一个工作流，以满足自己的需求。

## 1.1 软件开发者的工作流和任务清单

下面是一个典型的软件开发工作流，从中可以清楚地看到开发者编程时会经历的三个阶段（见图 1-1）。

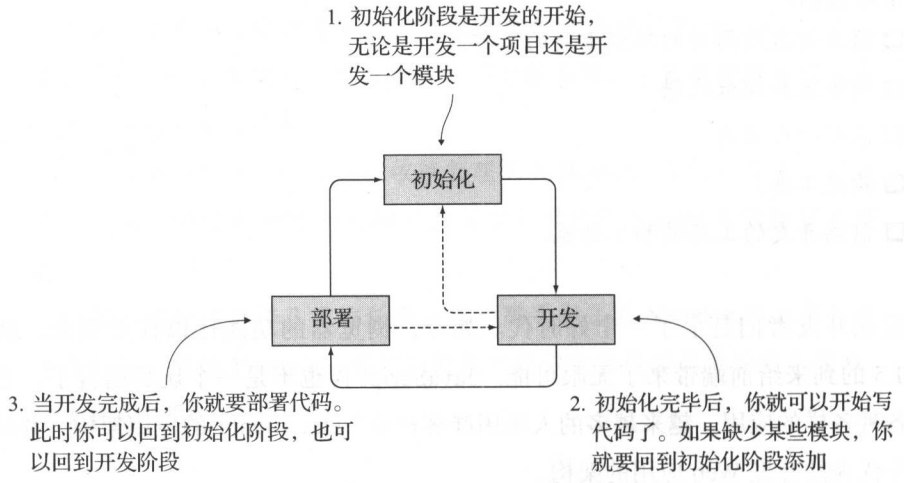



图 1-1 开发工作流的三个阶段

- 初始化：在任何软件开发过程中，初始化都是第一步。这一阶段的任务是搭建项目或者向已有项目中添加新文件。
- 开发：这一阶段是写代码的阶段。如果需要加载更多模块或者想把一些代码重构成一个新文件，那就得回到初始化阶段。如果一切都没问题，就能前进到下一阶段。
- 部署：到了这一阶段，代码已经写好了。这时就要创建可执行文件并且部署了。

对于 Web 开发而言，就是部署 HTML、JavaScript 和 CSS 到 Web 服务器。在这一阶段，开发者可以回到开发阶段（修复 bug 或者添加新功能），也可以回到初始化阶段（创建新模块）。

虽然所有软件开发流程都有上面这三个阶段，但是不同技术领域的开发者实现这些阶段中的任务的方法却是不同的。尽管其中有些任务非常单调乏味，而且和编程没有直接关系，但是却很大程度上影响了安装、架构和优化。图 1-2 展示了 JavaScript 项目中典型的任务。

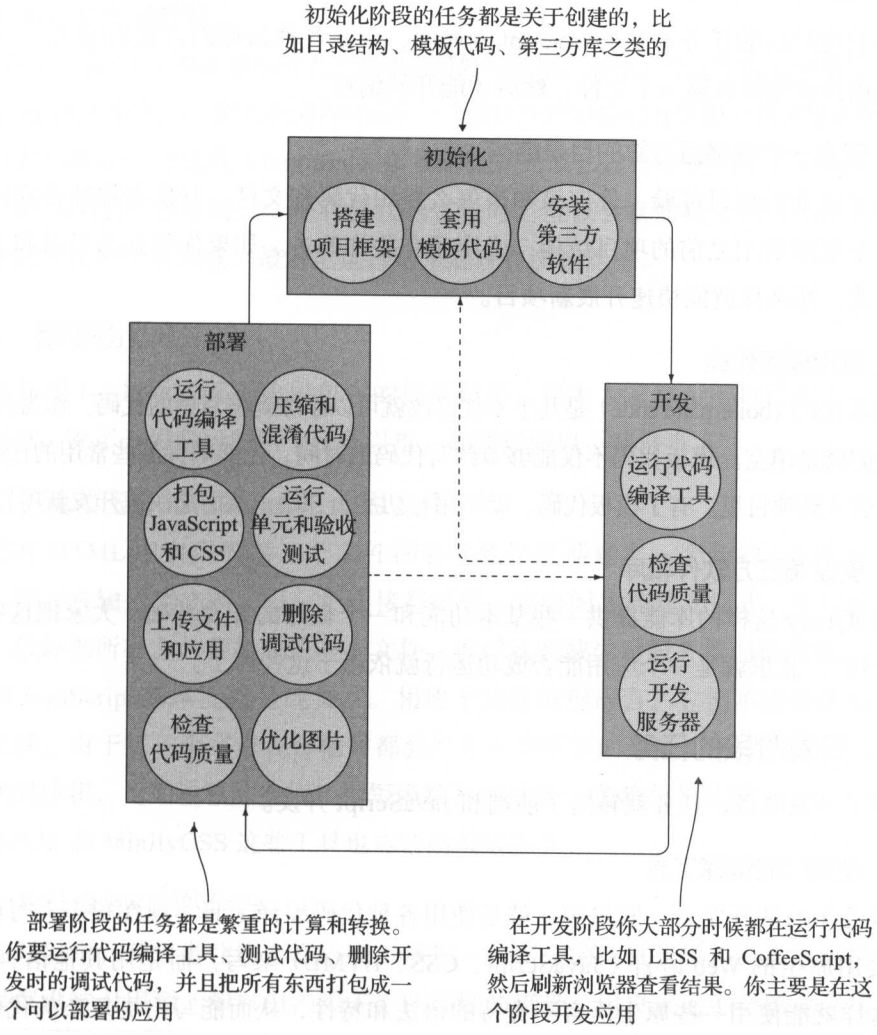



图 1-2　每个阶段的任务

初始化阶段的任务包括创建目录结构和应用模板代码，可能还会有下载和添加第三方库。开发阶段涉及一些代码编译工具，比如预处理器（LESS）或者 JavaScript 编译工具。大部分任务都是在开发阶段完成的。

图 1-2 展现了大量任务，接下来的章节会深入探究它们。读者接下来会面临很多任务，但是不要惊慌，本书的目的就是找到一个愉快而简单的自动化方案来解决它们。为了明白下一步要干什么，让我们逐个地来审视这些任务。

### 1.1.1 初始化阶段的任务

初始化阶段的任务不是 JavaScript 专属的，而是各种编程语言都适用的。一开始，你得创建目录结构和第一个文件，然后才能开始编程。

#### 1. 建立一个熟悉且可靠的目录结构

基于以前的项目经验，你应该知道怎么组织代码和文件。开发者往往会在接下来的项目中继续沿用之前的项目结构，或许还有些许改进。如果你很熟悉目录和文件的组织方式，那么你就能快速开展新项目。

#### 2. 复用模板代码

模板代码（boilerplate code）是几乎不加修改就可以添加到项目中的代码。作为开发者，你要做的就是填空。模板代码不仅能够节约写代码的时间，还能确保那些常用的代码都能正确地引入到项目里。有了模板代码，就不用一切从头开始，从而能快速开发新项目。

#### 3. 安装第三方软件和库

像 jQuery 这样的库能提供一些基本功能和一个良好的开发基础。大家把这些库叫作“依赖”，意思就是一个应用能否成功运行就依赖于这些库了。

### 1.1.2 开发阶段的任务

到了开发阶段，任务就转向了前端和 JavaScript 开发。

#### 1. 使用代码编译工具

近几年前端开发有一股热潮，就是使用各种代码编译工具。现在很多人写前端都不直接用原生的 Web 语言（JavaScript、CSS、HTML）来写，而是用其他语言来写。因为这样就能使用一些原生语言不支持的语法和特性，从而能写出更复杂的代码。但是浏览器不能直接解释这些语言，所以需要把代码编译成浏览器可理解、可运行的代

码。开发阶段和部署阶段都会用到代码编译工具。

#### 2. 代码质量管理

开发者所写的代码应该是高质量的代码。它们应该易读、易懂、不复杂，而且没有“代码异味”（code smell）[⊖]。你可以按照代码规范或者使用代码风格检查工具（code style checker）来帮助你写出高质量的代码。像 JSLint 和 JSCS 这些工具都能根据一系列可配置的规则来检查开发者是否写出了不合规范的代码，并且还能提出警告。开发阶段和部署阶段都包括代码质量管理的任务。

#### 3. 运行开发服务器

Web 应用运行在 Web 服务器上才能取得最好的效果。如果一个应用需要调用后端接口，并通过 AJAX 获取数据，那么在编程时运行一个模拟生产环境的开发服务器就很重要了。

开发阶段的任务迭代（iteration）必须能快速完成，这样才能马上看到代码修改的结果，而不用浪费很多时间。保存源代码文件时，应该会自动触发相应的代码编译工具，并进行代码质量检查，最后把运行结果展现在浏览器里。

### 1.1.3　部署阶段的任务

正如图 1-2 所示，部署阶段包含的任务最多，因为大家都希望自己部署的应用能完美运行。除了开发阶段的两个任务以外，你还得做以下几件事。

#### 1. 打包和压缩 JavaScript 和 CSS 文件

你在 HTML 中引用的每一个文件都会导致浏览器建立一个新的 HTTP 连接去下载它。而在传输数据之前，每一个连接都需要一定时间才能建立完成。所以在部署代码时，最好把所有文件都打包成一个文件，这样就能减少对服务器的请求数。HTML、CSS 和 JavaScript 基本上就是纯文本。相比于其他编程语言，它们不需要被编译成二进制文件。由于每一个字母在传输时都会产生一个字节的流量，所以需要用某些方式优化代码体积，比如说移除空格、缩短函数和变量名，或者本书提到的一些优化技巧。像 UglifyJS 和 MinifyCSS 这些工具也能帮助做这些事。

#### 2. 移除没有用到的代码

在开发中，开发者往往会在应用中添加一些多余的调试代码。结果最后这些代码在哪里都没被引用到，却白白占用了生产环境的流量。用户永远不会感觉到这些代码

⊖ 指那些语法和功能上没有问题，但是风格非常糟糕，而且以后可能导致问题的代码。

的存在，所以最好删掉它们。

3. 优化图片

图片在一个网页的体积中占据了相当大的部分。由于图片格式的缘故，图片文件本身储存了很多元信息。这些信息不但无法被浏览器解释，反而白白占据了图片的体积。像 ImageOptim 和 ImageMagick 这些工具就能处理并删除这些元信息。它们还使用了优化过的压缩算法，能够分别减少 JPEG 和 PNG 图片将近 10% 和 40% 的体积。

4. 执行单元测试和验收测试

单元测试确保模块的功能可以正常运行，而验收测试检查软件的输出结果能否满足特定需求。这两个测试可以保证应用在一般情况下不会崩溃。只有通过了这两个测试，应用才能被部署。

5. 上传应用

最后，开发者要把文件部署到 Web 服务器上，并确保外网能够访问服务器。

如你所见，你有一大堆任务要做。除了要牢记这些任务以外，你还要处理一个可能会导致问题的额外情况，即人为因素。

### 1.1.4 人为因素

很多任务都可以借助正确的工具来完成。比如用代码编译工具就可以编译代码，在命令行里输几个命令就能测试代码质量。但问题是，开发者不仅要知道这些工具，还必须知道如何使用它们。

这个过程会很单调，而且非常容易出问题。你可能会忘了执行某个任务，又有可能在运行命令时漏了一个参数。特别是在团队协作时，问题会更加复杂。比如说，你怎么才能确保同事在写完代码后没有忘记压缩代码呢？

流程越复杂，就越有可能出问题。设想这样一个情况：你要在不同的阶段运行十个工具，还要把每个工具产生的结果传递给下一个工具，你很容易就会搞不清现在进行到哪一步了。事实上，你希望运行一个简单的命令 deploy 就能做到这些事。而且最好不仅是你自己，连你的同事，甚至机器都能运行这条命令。

再仔细想想，如果你希望在部署阶段通过一条 deploy 命令就能执行所有的任务，那为什么不干脆通过一条 init 命令来初始化应用呢？又或者是用一条 develop 命令来运行所有开发阶段的任务？之后你就会发现，你的愿望其实很容易实现。

## 1.2　JavaScript 工具和新的工作流

为了预防潜在的错误，开发者要尽量减少可能出错的地方。回顾你在项目中做的编程以外的任务，你会发现这些任务都只不过是在不同阶段处理代码而已。让我们回过头来看工作流的示意图，但这次我们关注的不是各个阶段本身，而是在这些阶段中代码发生了什么变化。如图 1-3 所示，图片的边框发生了一些变化。

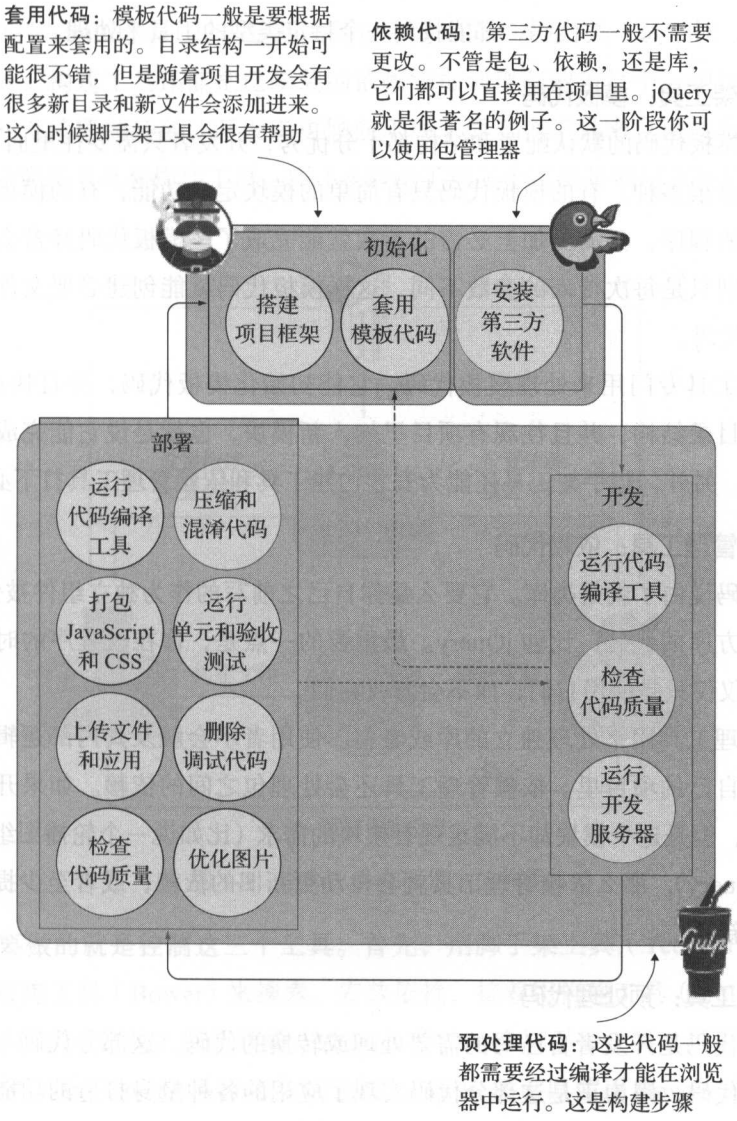



图 1-3　根据任务对代码进行分类

最后会产生三类代码：套用代码（模板代码）、依赖代码（库、依赖）和预处理代码（自己写的 JavaScript、CSS、HTML）。这三种类型的代码分别对应三种工具：脚手架工具、依赖管理工具和构建工具。让我们仔细研究一下这三类代码。

### 1.2.1 三类代码及其工具

如图 1-3 所示，根据每个任务所要处理的代码的类型，可以把任务分成三类。结果你会发现，对于每一类代码，都能找到一个特定类型的工具来处理。

#### 1. 脚手架工具：模板代码

典型的模板代码的默认配置和结构都十分优秀，开发者只需要往里面填空就行了。模板代码也有很多种。有的模板代码只有简单的模块定义功能，有的模板代码几乎就是一个完整的程序，只需要加上必需的参数就能完成了。模板代码经常会被复制到新项目中，区别只是每次传入的参数不同，这样模板代码就能创建必要文件让开发者能插入自己的代码。

脚手架工具专门用来处理模板代码。它能初始化模板代码，并且快速启动应用。它还能创建目录结构，并且往现有项目里加入新模块，也就是说它能完成两个初始化阶段的任务。另外，脚手架工具还能为安装构建工具和依赖管理工具打下必要的基础。

#### 2. 依赖管理工具：依赖代码

依赖代码说白了就是类库。它要么是你自己之前写的作为独立组件被使用的代码，要么是第三方库的代码，比如 jQuery。最重要的一点是，你在写程序的时候不会动这些代码。你仅仅只是使用它们，而不会修改它们。

依赖管理工具用来处理独立的库或者包。使用者不会触及其内部逻辑，而只是把它们添加到自己的项目里。依赖管理工具还会处理包之间的依赖。如果开发者想要安装一个模块，但是这个模块却不满足现有模块的需求（比如说一个轮播图组件需要一个新版本的 jQuery），那么依赖管理工具就会自动更新旧的依赖，或者至少提示开发者有依赖需要更新。

#### 3. 构建工具：预处理代码

预处理代码是开发者自己写的需要处理或转换的代码。这部分代码一般是开发者耗时最多的代码，因为正是这部分代码实现了应用的各种量身打造的功能。你可能常常听说这句话：“你写的代码不是你部署的代码。”因为开发者往往会使用 LESS 这样

的处理语言，并且打包 JavaScript 代码。另外，开发者还得注意代码风格并执行代码检查工具（lingting tool）。当你准备好部署代码时，还要压缩图片到最小体积，删除所有不需要的调试代码，删除没有用的 CSS，并且尽量减少文件数量。最后才能把所有的文件通过 FTP 上传到服务器或者 CDN。

构建工具专门用来应对所有的文件处理任务，把源代码转换成可以部署的代码。检查代码、编译、打包，这些任务构建工具也都能够处理。构建工具的配置一般保存在构建配置文件（build file）里，这些文件很容易修改和扩展。

除此以外，构建工具还能把已安装的依赖添加到现存的项目中。相比于用一大堆工具来处理开发阶段的任务，开发者更倾向于用下面三个工具来自动执行任务：脚手架工具、依赖管理工具和构建工具。图 1-4 展示了这三个工具是如何交互的。

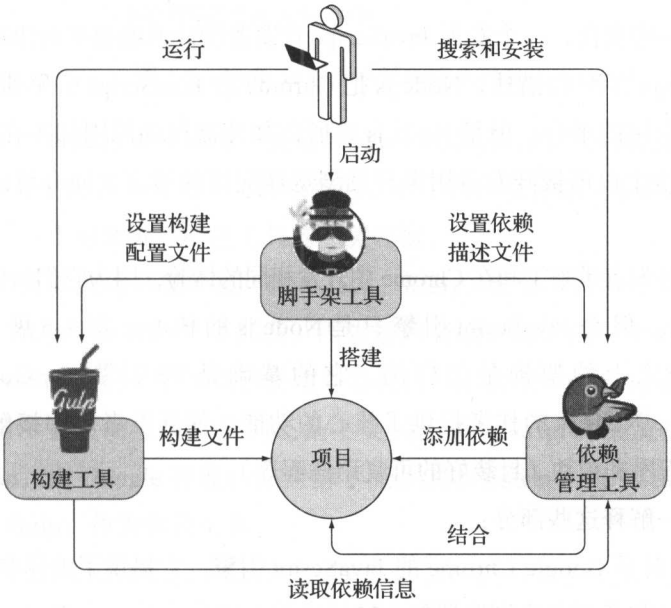



图 1-4　三个工具在项目中的作用

开发者要做的就是控制这三个工具。首先，用脚手架工具（Yeoman）搭建项目，然后用依赖管理工具（Bower）来搜索、安装依赖，接着用构建工具（Gulp）来处理文件。图 1-4 还展示了这些工具的联系。Yeoman 作为一个脚手架工具，可以初始化 Gulp（构建配置文件）和 Bower（项目基本的依赖）。Gulp 还可以使用 Bower 所提供的信息把依赖添加到项目的主要文件中。

项目本身是这三个工具的核心。开发者通过工具的接口把工具整合到项目中。开发者通过脚手架工具来创建新应用的基础架构，通过依赖管理工具搜索和添加依赖，通过构建工具运行一条命令把源代码和前两个工具创建的代码集成到项目里。这样依赖就能用三个简单的接口来封装这些复杂的任务，让工具来处理最复杂的工作。迄今为止，很多运行时和运行环境都提供了这些工具。不过随着 Node.js 的出现，开发者有了一个更适合 JavaScript 的新环境。

### 1.2.2 Node.js 中的 JavaScript 工具

因为现在这三种工具每一种都有很多选择，所以决定使用哪一种工具的责任就落到了开发者头上。以前，尽管所有处理任务的工具都是用 JavaScript 写的（也有些著名工具是用 Ruby 写的），但是集成工具却都是用和 JavaScript 完全无关的语言写的。但是近几年发生了一些变化，一个专为 JavaScript 开发者设计的编程平台出现了：Node.js。

根据 Node.js 官网的描述，Node.js 把 Chrome 的 JavaScript 引擎带到了一个适合开发快速网络应用的平台。但是 Node.js 之所以如此流行的原因却不在于此，而在于：JavaScript——这个以前被开发者用来开发浏览器应用的语言，如今可以摆脱浏览器的束缚了。

Node.js 给开发者带来了和在 Chrome 中开发相同的体验，因为它们所使用的 JavaScript 引擎是一样的。但是 JavaScript 引擎只是 Node.js 的核心，图 1-5 展示了 Node.js 在 JavaScript 引擎之上的架构是怎样的。它的基础是 V8 引擎——Google Chrome 的 JavaScript 引擎。Node.js 的作者提供了核心的功能，使开发者能够接触操作系统，最上面的一层是程序和模块（封装好的可复用的程序）。

让我们逐一解释这些部分：

- V8 运行时是 Google Chrome 的 JavaScript 引擎。它提供了编译和运行 JavaScript 的环境，但是不包含浏览器的 API。
- Node.js 核心由和操作系统相关的 API 组成，包括文件系统、网络协议和流，简而言之就是所有涉及 I/O 和操作系统的功能。
- 程序是可以在 Node.js 中运行的 JavaScript 代码。
- 模块是 Node.js 中的一个概念，用来保证代码的复用性。所有的程序都可以被打包成一个模块，并且作为一个包发布。模块提供了封装好的功能，开发者可以在新程序中复用它们。

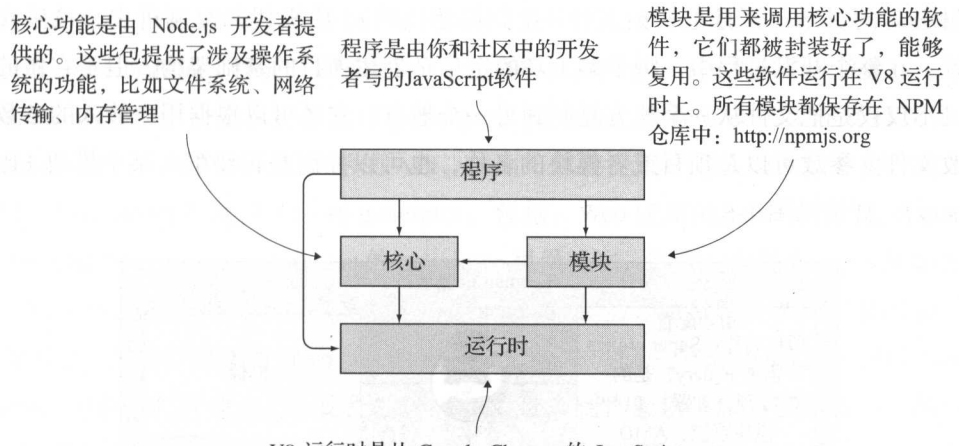



图 1-5　Node.js 生态系统的组成部分

对工具而言，Node.js 最重要的一点是 Node.js 核心层。因为 Node.js 提供了操作系统相关的功能，其中大部分是关于文件系统的。有了操作文件系统和修改文件的能力，Node.js 就成为了一个用来编写构建工具的完美环境。

Node.js 有一点和其他开发环境（比如 Java、Ruby）完全不同：开发者不需要任何心理负担就能从前端开发转向工具开发。因为这两者用的都是同一个语言，运行环境也是一样的。JavaScript 开发者可以轻松地开发自己的工具用来转换 JavaScript 代码。

尽管还有很多替代的工具，我们还是主要关注 JavaScript 界最流行的三个工具，分别是：Yeoman，由 Google 开发，作为脚手架工具；Bower，由 Twitter 开发，作为依赖管理工具；Gulp，作为构建工具。



注意　之后所有在命令行中运行的示例代码都是根据 Unix bash 语法而写的。但这并不意味着这些代码只能在 Unix 系统中运行。Unix/Linux bash、Mac OS X 终端和 Windows 的命令提示符都能以同样的方式运行这些代码。

## 1.3　用 Yeoman 搭建脚手架

脚手架工具的作用是搭建项目，让项目顺利跑起来。脚手架工具可以创建一个

项目必需的目录，并且把初始文件（比如构建代码）、模板代码复制进去，最后安装依赖。在整个开发过程中，脚手架工具的定位是创建项目的基础架构。但是它做的事情又不仅仅是把文件从一个地方复制到另一个地方。它还可以根据用户输入的参数而修改文件。参数可以是项目或者模块的名称，也可以控制是否要引入某个模块（比如 jQuery）。详情见图 1-6。

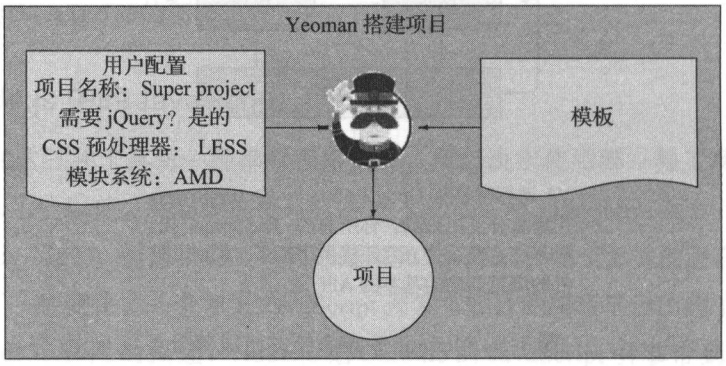



图 1-6 Yeoman 会从 generator 接受定义好的模板，然后根据用户输入的参数，对模板进行一些修改。修改好的模板可以被添加到现有项目中，也可以用来创建一个全新项目

### 1.3.1 Yeoman 作为脚手架工具的优点

搭建脚手架可以通过机器完成，而且通常它是 IDE 的一个功能。在像 Eclipse、WebStorm 和 IntelliJ 这类工具中，你可以用模板创建一个新项目，或者创建一个新模块。通常还会有一个向导来向你询问一些搭建脚手架所必需的参数。

如果你选择用 IDE 来搭建脚手架，你就得在开发过程中一直用它，间接导致你的同事也不得不使用它。这种时候，像 Yeoman 这样的独立脚手架工具就有巨大的优势：它的功能很单一，就是启动一个项目和模块，和其他开发环境并不耦合。因此，无论你的环境或者工具链怎么改变，都可以使用 Yeoman 来启动新项目。即使你在项目开发到一半时放弃 Yeoman，也不会给你的项目带来什么问题。而且 Yeoman 可以像很多其他 Node.js 工具一样在命令行运行，这和 Bower 和 Gulp 是一样的。

但是 Yeoman 最大的好处在于它所使用的名为 generator 的插件系统。

### 1.3.2 什么是 generator

generator 是可以被命令行工具 yo 执行的 JavaScript 应用，它所做的事情只不过是创建

新目录和复制文件。不过，它也没有那么简单。它还可以监听参数来修改文件。generator 一般由一个目录构成，目录中是模板文件，这些模板会被 generator 的代码所修改。

Yeoman 的唯一任务就是运行 generator。generator 知道应该把文件复制到哪里。generator 有一个简单的安装提示的 API，从而可以接受用户输入的参数并更改输出的结果。Yeoman 团队制作了一些 generator，包括：Web 应用的 generator，还有 Ember 和 Angular 的 generator。Angular 的 generator 很受关注，因为它也是 Google 开发的。

Yeoman 用起来很简单。你安装完一个 generator 以后，就可以在命令行里用 yo 命令来调用它。你可以直接通过 yo generator-name 命令来执行一个 generator。下面以 Angular generator 为例。打开命令行（或者终端），然后进入到你想创建新项目的目录。执行 yo angular 命令来让 Yeoman 启动 Angular generator。你会看到一些安装提示，需要你回答问题：

```
$ yo angular                                    ❶ 启动 Yeoman 的命令
Out of the box I include Bootstrap and some AngularJS recommended modules.
? Would you like to use Sass (with Compass)? Yes
? Would you like to include Bootstrap? Yes
? Would you like to use the Sass version of Bootstrap? Yes
? Which modules would you like to include? (Press <space> to select)
?? angular-animate.js
 ? angular-aria.js
 ? angular-cookies.js
 ? angular-resource.js
 ? angular-messages.js
 ? angular-route.js
```

❷ 大多数问题是非题

还有一些进阶的问题 ❸

在 shell 或者命令行中执行这个命令来启动 Yeoman ❶。这个命令由 yo 和 generator 的名字组成。根据你的回答，generator 会决定是否引入相应的代码❷。其中有一个多选题，里面有一些预先选好的推荐模块。你也可以用空格来取消它们❸。

通过 Yeoman 你很快就可以启动一个新项目。就比如说这个 Angular 项目，它已经被很多开发者检验过，成为了社区里的标准，你完全可以相信它。这样你就不用把时间浪费在目录、文件结构、名称这些细节上，可以更加专注于开发。

下面让我们看看依赖管理工具。

## 1.4　依赖管理工具 Bower

依赖是独立完整的模块，它是一个应用的一部分或者扩展。依赖可以是像 jQuery 或 Angular 这样的库，可以是一些 UI 组件，也可以是像 Bootstrap 这样完整的 UI 框

架。依赖的另一种叫法是包（package）。使用依赖或者包的好处在于：你不用花时间去开发那些已经有的功能，只要用现成的就行了。

依赖管理工具的主要功能是在社区和公司的仓库中搜索依赖，并且下载到本地的项目中。由于包也可以拥有依赖，因此依赖管理工具的任务绝不仅仅只是把文件下载下来而已。如果有两个或以上的包都依赖 jQuery，但是依赖的版本却不同，那么依赖管理工具此时就要解决这个冲突。解决方法可以是选择其中一个，也可以两个都安装，或者是让用户来决定选择哪一个。参看图 1-7。

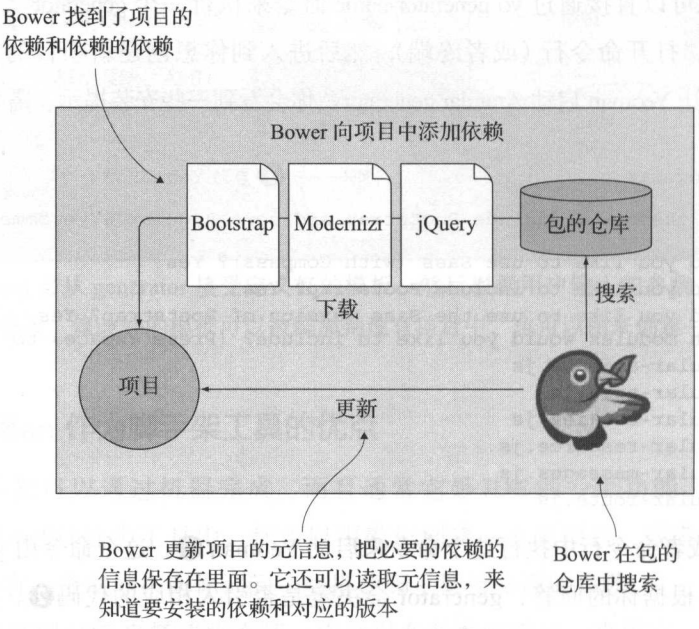



图 1-7 Bower 作为包管理器能够在包的仓库中搜索依赖。如果发现版本不符合，它还会更新依赖。如果一些正常，这些依赖就会被下载下面并添加到项目里

### 1.4.1 Bower 的优点

Bower 自称是为 Web 而生的包管理器，事实上也是这样。它是由 Twitter 开发的，专门用来从网上安装框架、库和其他模块。只要一个模块是由 Git 托管的，并且拥有 bower.json 文件，就可以用 Bower 来安装。

Bower 是专门为前端开发者而设计的。但是这并不是说它只能引入 JavaScript、CSS 或者其他相关的语言。但是由于它是为前端而开发的，而且所有的包都需要在 HTML 引入才能使用，所以其他语言的开发者也不会用它。Bower 提供了搜索、安装

和更新包的命令。当你要安装一个包的时候，Bower 会检查这个包需不需要其他的包（比如 jQuery）。而且，它还会检查是否有哪个包重复安装了，或者版本冲突了。如果没有冲突的话，那么新的包就会被添加到项目中。

在众多包管理器中，我们选择了 Bower。理由主要是下面几条：

- Bower 是为前端开发者设计的。所有主流的前端库都能够使用这个工具。
- 包的仓库（repository）的发展非常迅速。由于 Bower 专注于前端领域，所以包的数量都还可控，也不会遗漏哪个包。
- Bower 的依赖树是扁平的。Bower 的这一点不仅是独一无二的，而且对于 Web 开发者也是必要的。实践表明，扁平的依赖树的结构是最适合 Web 应用的，因为你不会（至少你不需要）安装两个同样的包。

### 1.4.2　Bower 的依赖树

依赖树是用来展示依赖之间的关系的。以 Bootstrap 为例，它需要 jQuery 才能运行。在依赖树中，jQuery 是 Bootstrap 的一个子节点。如果你安装了另一个依赖 jQuery 的组件，你有两个选择：

- 再次下载 jQuery，然后把 jQuery 作为这个新的组件的子节点。
- 把这个新组件引用的 jQuery 指向已经安装的 jQuery。

图 1-8 展示了这两个选择。

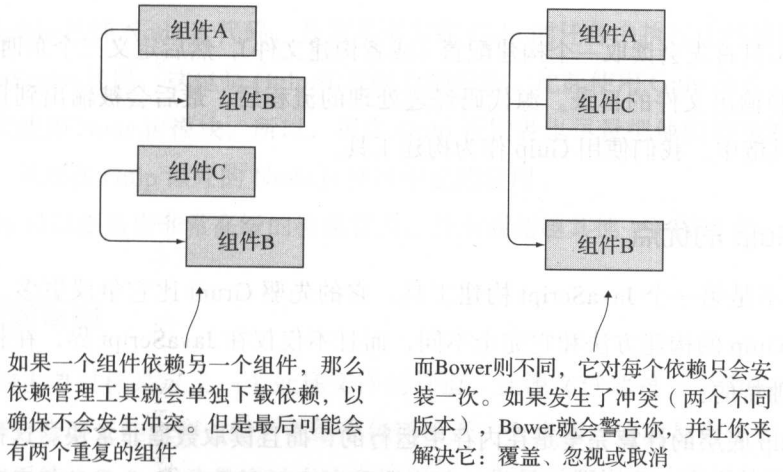



图 1-8　这是两种依赖树：一个是深依赖树，它会重复安装依赖；另一个是扁平依赖树，每个依赖只会安装一次

这两个选择都有优缺点。深依赖树可以确保安装的依赖完美兼容组件，依赖的API就是组件需要的API。但是每次都重新安装依赖，会导致重复安装。另一个选择，扁平依赖树则可以确保每个依赖只会安装一次，整个项目里每个组件只有一个，但是这可能会导致依赖之间的版本冲突。

Bower使用了后者，理由是：在Web应用中，不应该把一个包重复传输好几次。如果你安装了jQuery，并且在整个项目中使用，那么浏览器就没办法在另一个地方再运行一个jQuery了。扁平依赖树更适合前端开发。如果安装依赖的时候出现了冲突，它会警告开发者，此时开发者可以采取下面这些选择：

- 取消安装
- 用新的依赖覆盖旧的
- 仍然使用旧的依赖

有了Bower，你就可以用工具来管理所有的第三方模块和组件了。如果你选择第三个选项，就可以保证依赖和代码都可以正常工作。

## 1.5 Gulp流式构建系统

构建系统是整个工具链的核心。它会处理所有的源文件，把它们变成可以部署的代码。它还可以检查代码质量，自动执行重复的操作。图1-9展示了构建工具是如何工作的。

构建工具首先会读取一个构建配置（或者构建文件），然后定义三个东西：源文件、处理流程和输出文件的目录。源代码经过处理的流程后，最后会被输出到目录中。在我们的工具链中，我们使用Gulp作为构建工具。

### 1.5.1 Gulp的优点

Gulp不是第一个JavaScript构建工具。它的先驱Grunt比它争议更多，历史也更久。但是Gulp的构建方法和它完全不同，而且不仅仅在JavaScript界，在整个构建系统界都是独特的：

- Gulp底层的计算完全是在内存中运行的，而且读取数据非常快。这样就可以快速转换文件，灵活组合构建管道。如果构建管道里有一个工具不需要了，只要把调用它的那行代码去掉就行了，其他部分仍然能照常运行。

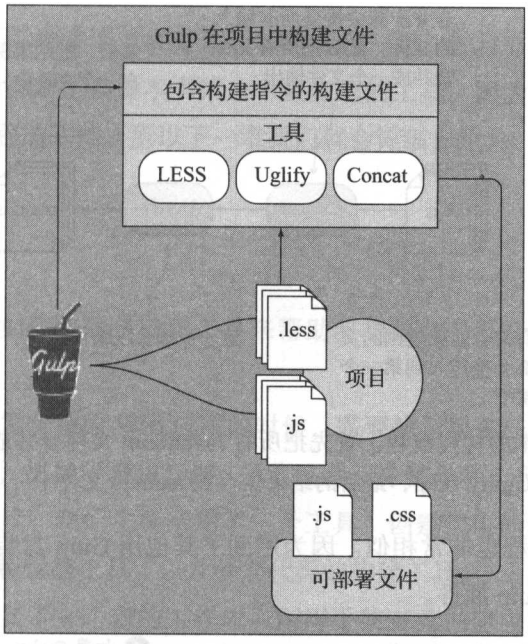



图 1-9　Gulp 作为一个构建工具，可以调用其他工具来把源代码转换成可部署的文件，构建的命令都保存在构建文件中。Gulp 工作的流程是：接受输入的文件，对它们执行不同的命令（或工具），然后把结果保存在新目录中

- Gulp 可以并行执行任务。开发者可以指定哪些任务按顺序运行，哪些同时运行。比起其他工具，这又是一个性能提升。
- Gulpfile 是纯 Node.js 代码，开发者可以在 Gulp 的任务中执行任何兼容 Node 的 JavaScript 代码，这也是 Gulp 作者极力推崇的。比起使用 Gulp 插件，它更青睐直接使用 Node.js 模块。所以，很多 Gulp 开发者也尽量把他们的功能做得更通用，从而在 Gulp 以外的 Node.js 社区中也能使用。
- Gulp 可以创造出非常高级的构建管道，甚至能超越其他一流的工具。

### 1.5.2　构建管道

Gulp 的主要目标是定义一个处理文件的管道，让源文件通过一个个的处理步骤。每个步骤都会修改文件内容，然后把结果传递给下一个步骤。这就是为什么 Gulp 自称是流式构建系统。Gulp 插件是通过方法来调用的，你只要调用它，剩下的工作都由插件来完成。图 1-10 展示了这个过程。

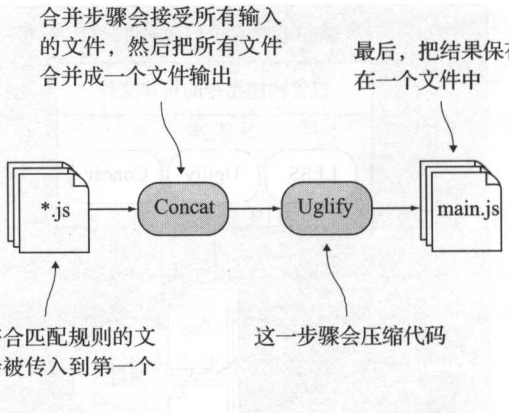



图 1-10 基本的构建流程。首先把所有 JavaScript 文件合并成一个，然后用 Uglify 压缩，最后的结果保存在 main.js 文件中

实现的代码和流程也非常相似。因为用到了其他用 Gulp 封装的工具，所以你要首先加载其他必要的 Gulp 插件。

```
var gulp = require('gulp'),                    <-- ❶ 加载 Gulp
  uglify = require('gulp-uglify'),             <-- ❷ 加载 Uglify 插件
  concat = require('gulp-concat');             <-- ❸ 加载 Concat 插件
```

首先要加载 Gulp ❶。Uglify 插件包括了原始的 Uglify 程序，它是用来压缩 JavaScript 代码的，同时也提供了很方便的 API ❷。要合并文件可以使用 Concat 插件。这个插件是 Gulp 作者写的，因为这个需求太普遍了❸。

然后要定义你要执行的 task。下面的代码就是图 1-10 的实现。

```
gulp.task('scripts', function() {          <-- 定义你的 task，并且把它命名为 script
    return gulp.src('src/**/*.js')         <-- 选择所有 src 目录中的 JavaScript 文件
        .pipe(concat('bundle.js'))         <-- 合并成一个文件
        .pipe(uglify())                    <-- 执行压缩步骤
        .pipe(gulp.dest('dest'))           <-- 保存结果
})
```

整个处理过程都是在内存中进行的，只有在最后才会把结果写入硬盘中。这使得整个过程非常迅速。自从 2013 年 Gulp 发布以来，Gulp 就一直受到对自动化和性能有着强烈需求的开发者们的追捧。

Gulp 也是从命令行执行的。只要在 bash 或者 shell 中执行 gulp scripts 命令，Gulp 就会执行定义好的 scripts task。然后 Gulp 就会从 Gulpfile 读取构建指令，并且把它传给 Node.js 的运行时环境。

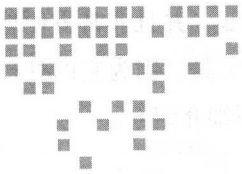
通过 Gulp，你可以把多个文件处理工具整合在一起，并且非常方便地在命令行里调用它们。Gulp 作为构建工具是本书第一个要介绍的工具。因为它的处理文件的功能是整个项目中所有流程的基础，所以下一章我们就会详细介绍它。

## 1.6　总结

本章主要介绍了如何使用自动化工具来帮助前端开发者更好、更快地管理复杂的任务：

- 前端开发者开发复杂的 Web 应用的时候，需要执行很多任务，其中包括繁重的文件转换任务，比如压缩和打包。所有这些任务都是和工具紧密联系在一起的。
- 为了降低复杂度，我们主要采用了三个工具：创建代码的脚手架工具、下载和安装代码的依赖管理工具，以及处理代码的构建工具。
- 这三个工具定义了一个新的工作流：用脚手架工具创建应用，用依赖管理工具下载需要的依赖，然后使用构建工具来执行代码编译工具。
- 这三个工具都有基于 Node.js 的版本。Yeoman 是 Google 开发的脚手架工具。它特有的 generator 件可以根据不同的设置来搭建脚手架。
- Bower 作为依赖管理工具可以检查模块和组件的版本，并它们集成到项目中。它采用了扁平依赖树，可以避免重复的依赖，这一点对 Web 应用非常合适。
- 构建工具 Gulp 有非常方便的命令行接口，还有众多插件可供使用。只需要几行代码，就能用链式操作把文件转换步骤连接起来。

Gulp 是整个项目的核心，也是最主要的工具。第 2 章你就会开始学习使用 Gulp。



Chapter 2 第 2 章

# Gulp 入门

**本章包括：**

- Gulp 的介绍。
- 流的概念。
- 给自动化工具创造简单任务。
- 为多功能任务创建执行管道。

创建新工作流时，最重要的是构建系统。虽然构建系统在上一章是放在最后介绍的，但是它却是整个工作流必要的基础。构建系统的作用是处理日常工作流产生的任务，比如合并、压缩和测试代码。构建工具是由脚手架工具来初始化的，然后构建工具会把依赖管理工具集成到项目中。图 2-1 展示了构建工具在整个工作流中的定位。

Gulp 是实现自动化工作流的基础，它是脚手架工具最先初始化的元素，之后构建工具再把其他组件集成到项目中。而且，由于构建工具能创建开发环境，所以它在整个开发周期中也是最常用的工具之一。

Gulp 是用 JavaScript 编写、运行在 Node.js 上的构建工具。因为 Gulp 是一个 JavaScript 程序，而且它的指令也是用 JavaScript 编写的，所以 Gulp 非常贴近 JavaScript 开发者的日常工作环境。因此，Gulp 就成了前端开发者自动化处理日常任务的首选工具。这一章讲的是如何为项目配置 Gulp，从中你会了解到 Gulp 配置的方方面面，并且学会

如何编写构建指令来构建 JavaScript 和 CSS。

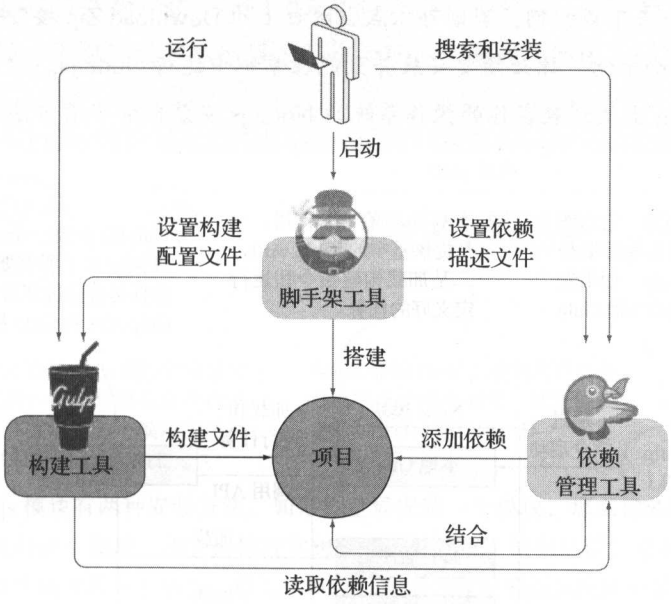



图 2-1　第 1 章的工作流配置

## 2.1　配置 Gulp

作为一个 Node.js 工具，Gulp 是由很多部分组成的。这一节会介绍 Gulp 的各种构建块以及如何安装 Gulp 的命令行接口和本地 Gulp。

### 2.1.1　Gulp 的构建块

一般而言，构建工具由两部分组成：执行构建任务的工具和包含构建指令的配置文件。Gulp 也不例外。Gulp 执行构建任务的工具就是 Gulp，而 Gulp 的构建配置文件叫作 Gulpfile。Gulp 还能被继续拆分成其他几个部分。图 2-2 展示了 Gulp 的各个部分以及它们之间的关系。

**项目示例和 Node.js**

本书提供了所有可能用到的项目示例，还准备了一个示例程序，读者可以在

http://github.com/frontend-tooling/sample-project-gulp 查看。如果你熟悉 Git，那你可以使用 clone 命令来下载它们，否则可以点击网页上的 Download Zip 按钮来下载文件。

要使用这些示例，你必须要安装并且能在系统中运行 Node.js。你可以在 https://www.nodejs.org 上找到兼容你的操作系统的 Node.js 安装包和安装方法。

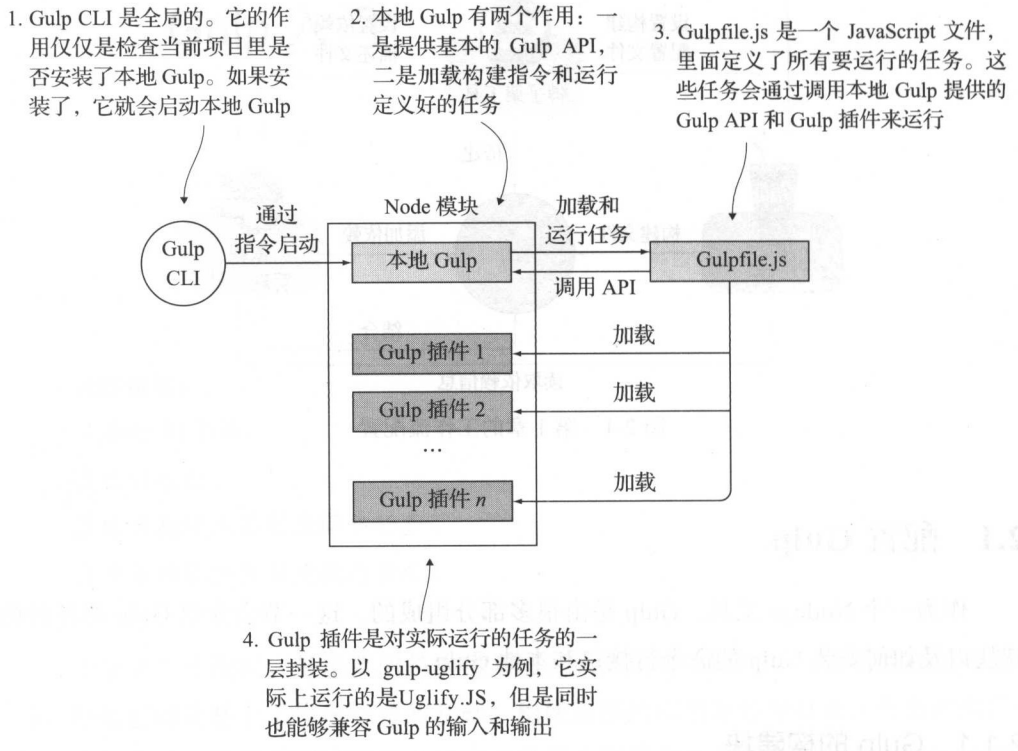



图 2-2 Gulp CLI 根据命令行中输入的参数启动本地 Gulp。本地 Gulp 读取 Gulpfile。Gulpfile 加载 Gulp 插件，并且用 Gulp API 定义 task。最后由本地 Gulp 来加载这些 task

完整的 Gulp 安装流程包括以下几个部分：

- Gulp CLI：启动构建工具的命令行接口。
- 本地 Gulp：构建时实际运行的程序。
- Gulpfile：告诉 Gulp 如何构建软件的指令文件。
- Gulp 插件：众多用来合并、修改、组装文件的插件。

要做的事情真多啊！但是不用担心，这一章就能把这些事全搞定。首先从 CLI 开始。

### 2.1.2　Gulp 命令行接口

Node.js 模块可以全局安装，所以用户可以像使用命令行工具一样使用 Node.js 模块。Node.js 模块也可以安装到本地目录，作为项目依赖的库使用。图 2-3 展示了这两种安装方式。

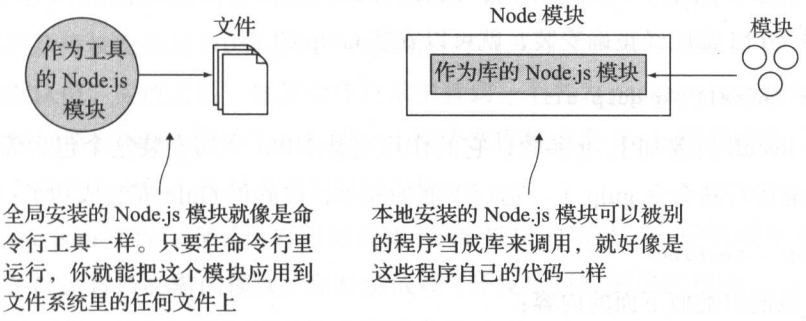



图 2-3　Node.js 模块有两种安装方法。如果你全局安装一个模块，那么你就能在命令行里直接执行这个模块，用它来修改你想要修改的文件。换句话说，它就是一个工具。如果你本地安装一个插件，那么你就可以在别的程序里面调用这个模块的功能

Gulp 也是如此，除了有一点不同，那就是 Gulp 有两个独立的包。一个是 Gulp 的命令行接口（以下简称 Gulp CLI），它需要全局安装，所以 Gulp 能够在终端、bash 和命令提示符中直接运行。另一个是本地的 Gulp，它是真正的 Gulp 运行时。负责处理所有任务，并且提供所有的 API。Gulp CLI 是本地 Gulp 的全局的入口，它负责把所有参数转发到本地 Gulp，还有显示项目里安装的本地 Gulp 的版本。后面的就都是本地 Gulp 的任务了，主要由它来处理构建。

Gulp CLI 看起来有点傻，因为它的唯一作用就是检查能否运行本地 Gulp。这么设计的原因是为了确保即使 Gulp CLI 和本地 Gulp 的版本不同，也能够运行 Gulp。想象一下，你有一个项目用的 Gulp 是 3.8 版本的，而 Gulp CLI 仍然能运行这项目，原因就是本地 Gulp 的接口是一样的。即使是一个用了 Gulp 4 的新项目也能用同一个 Gulp CLI 运行。你会发现，无论本地 Gulp 是什么版本的，你都能用同一个 Gulp CLI 运行，而且甚至你可能还从来没更新过 Gulp CLI。

要安装 Gulp CLI，首先启动 bash、终端或 Windows 的命令提示符（这些平台都支持），并且输入下面的命令，检查是否正确安装了 Node.js：

```
$ node --version
```

如果已经安装了 Node.js，这行命令就会输出当前 Node.js 的版本。下一步是检查 Node.js 的包管理器——NPM，它是随着 Node.js 一起安装的。运行下面的命令：

```
$ npm --version
```

同样，这行命令的输出结果应该是 NPM 的版本。如果这两个程序中有一个没有正确安装，请参考附录中的安装指南，或者访问 Node.js 的官网 https://nodejs.org。如果 Node.js 和 NPM 都已经正确安装，就可以安装 Gulp CLI 了：

```
$ npm install -g gulp-cli
```

注意 install 后要加上 -g 参数，它的作用是让 NPM 全局安装这个包。等到安装完成后，就能运行新命令 gulp 了。运行下面的命令，以确保 Gulp 安装成功了：

```
$ gulp --version
```

它应该输出类似下面的内容：

```
[12:04:15] CLI version 0.2.0
```

到此，安装 Gulp 的第一步就完成了，你已经成功地在你的系统里安装了 GulpCLI！接下来，让我们继续安装本地 Gulp。

### 2.1.3 安装本地 Gulp

本地 Gulp 有两个主要作用：一个是加载和运行 Gulpfile 中的构建指令，另一个是暴露 API 供 Gulpfile 使用。本地 Gulp 是真正运行构建任务的程序，全局 Gulp 只是用来启动各个项目中的本地 Gulp，图 2-4 展示了这种关系。

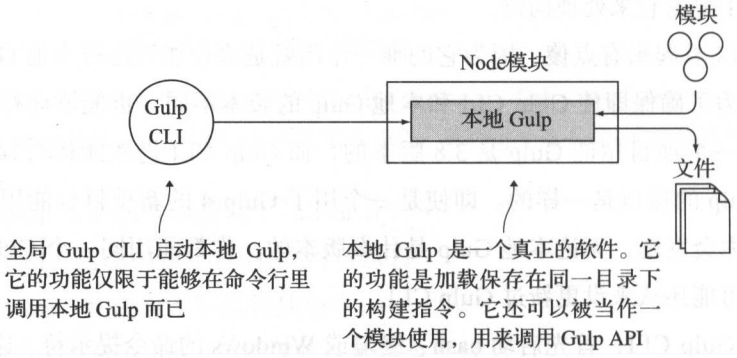



图 2-4 全局安装的 Gulp 是 Gulp CLI。它的作用是检查本地 Gulp 是否可用，如果可用就调用它。本地 Gulp 位于本地 JavaScript 项目的 node_modules 目录下，它包含了 Gulpfile 所需的所有函数和 API

要安装本地Gulp，首先打开命令行，并进入到示例项目的目录。不要直接进入到app文件夹里，而是进入到上一层，也就是README.md文件所在的目录中。然后，通过下面的命令，把整个项目初始化成一个Node模块。

```
$ npm init
```

输入命令后出现的是一个简短的安装提示，其中会询问一些项目信息。不过如果你不打算把新模块发布到NPM仓库（registry）中（或者至少现在不想），你可以把所有值都设为默认值。完成之后，你就会在目录中看到一个名为package.json的新文件，它保存了应用的所有Node模块信息和版本。这个文件是每个Node项目的核心，而且有的插件也会从这个文件中获取已安装的模块信息。

package.json文件保存了所有项目依赖Node模块的信息。它把依赖分为运行时依赖（项目正确运行所必需的模块）和开发依赖（开发项目时需要的模块）。因为构建工具属于开发依赖，所以要用下面的命令安装本地Gulp：

```
$ npm install --save-dev gulp
```

Gulp安装好了，save-dev参数的作用是把安装的Gulp的版本正确保存在packge.json文件中：

```
{
  "name": "sample-project-gulp",
  "version": "1.0.0",
  "description": "The sample project we will use throughout the Gulp
     chapters",
  ...
  "devDependencies": {
    "gulp": "^4.0.0"
  }
}
```

然后再次检查版本（用gulp --version命令），输出结果应该会发生变化。Gulp CLI将会识别出本地Gulp：

```
[20:40:12] CLI version 0.2.0
[20:40:12] Local version 4.0.0
```

### Gulp 4

本章以及本书的其他部分是根据最新的Gulp版本Gulp 4来写的。但是截至本书出版为止，Gulp 4仍然处于预发布阶段。所以我也不知道当你现在阅读本书时，

Gulp 4 是否正式发布了[⊖]。如果 gulp --version 给出的版本是 3.x，那么 Gulp 4 就仍处于预发布阶段。如果要安装 Gulp 4，就要用下面的命令安装：

```
$ npm install --save-dev gulpjs/gulp.git#4.0
```

这行命令会下载并安装预发布分支的 Gulp。

现在 Gulp CLI 和本地 Gulp 都安装好了。下一步是写 Gulpfile。

## 2.2 创建 Gulpfile

前几节里我们已经安装好了 Gulp。Gulp CLI 和本地 Gulp 都已经能够正常运行了。到目前为止，我们已经能用 Gulp CLI 启动本地 Gulp 了，详见图 2-4。接下来就要进入到下一环节 Gulpfile。

### 2.2.1 Gulp 的“Hello World”task

Gulpfile 是包含了所有安装指令的 JavaScript 文件，安装指令包含一系列被打包成 task（任务）的命令。在 Gulp 中，task 是一个 JavaScript 函数，其中包含了所有要运行的命令。task 可以是任何函数，但是只有要用 Gulp API 中的 gulp.task 方法来定义这个函数，它才是一个 task。图 2-5 展示了这个方法的用法。

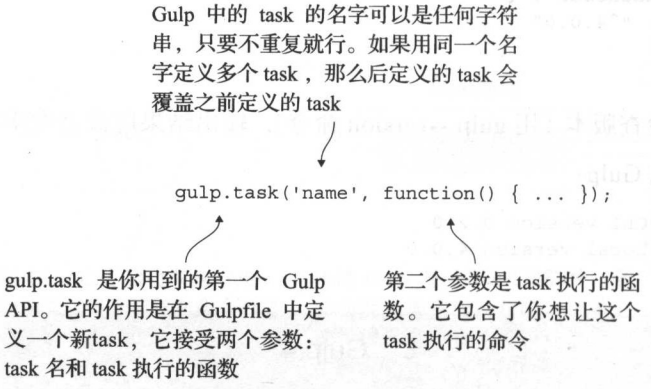



图 2-5 gulp.task 方法的用法说明

⊖ 目前 Gulp 4 仍然没有正式发布。——译者注

gulp.task方法的作用只有一个：给纯JavaScript构成的task函数赋予一个名字。这个名字会把task函数添加到Gulp的运行时的命名空间中。这样一来，Gulp就能知道这个函数的存在，并且能根据引用来执行它了。也就是说gulp.task可以提供一个在命令行中直接调用这个函数的接口。

首先让我们来写一个task。按照编程界的传统，我们就试着在命令行中输出一个“Hello World!”，首先在包含package.json和node_modules的目录中创建一个新的空文件，名称叫作Gulpfile.js。下面是要添加的文件内容：

**代码清单2-1 Gulp的Hello World——Gulpfile.js**

```
var gulp = require('gulp');          <-- ❶ 在Gulpfile中引用本地Gulp

gulp.task('test', function () {      <-- ❷ 定义一个名叫test的新task
  console.log('Hello World!');       <-- 在命令行打印“Hello World!”
});
```

在Gulpfile中引用了本地Gulp后，就能够调用Gulp API了❶。Gulp和Gulpfile之间是互相关联的：Gulp加载Gulpfile，这样Gulp才知道一共有哪些task；Gulpfile加载Gulp，这样才能使用Gulp API。定义了test task后，Gulp就能够执行这个task了。gulp.task的第二个参数，是需要执行的函数。

Gulp和Gulpfile之间有一种独特的关系：Gulp需要Gulpfile才能知道有哪些task可以运行；Gulpfile需要Gulp才能调用Gulp API。图2-6展示了其中的细节。

使用gulp.task这个API就能把一个函数，变成一个task。在代码清单2-1中创建的“Hello World!”task就是一个名为test的task。这个名字会保存在Gulp的运行环境中。你可以通过下面的命令，在调用Gulp命令时，直接执行这个task：

```
$ gulp test
```

执行上面这个task的具体流程（如图2-7所示）如下：

- 全局Gulp CLI加载本地Gulp。
- 本地Gulp加载Gulpfile。
- Gulpfile加载本地Gulp，然后定义一个名为test的task。
- 本地Gulp接收到一个命令行参数，也就是要执行的task的名字。
- 本地Gulp发现确实有一个task叫这个名字，于是执行了这个task所绑定的函数。

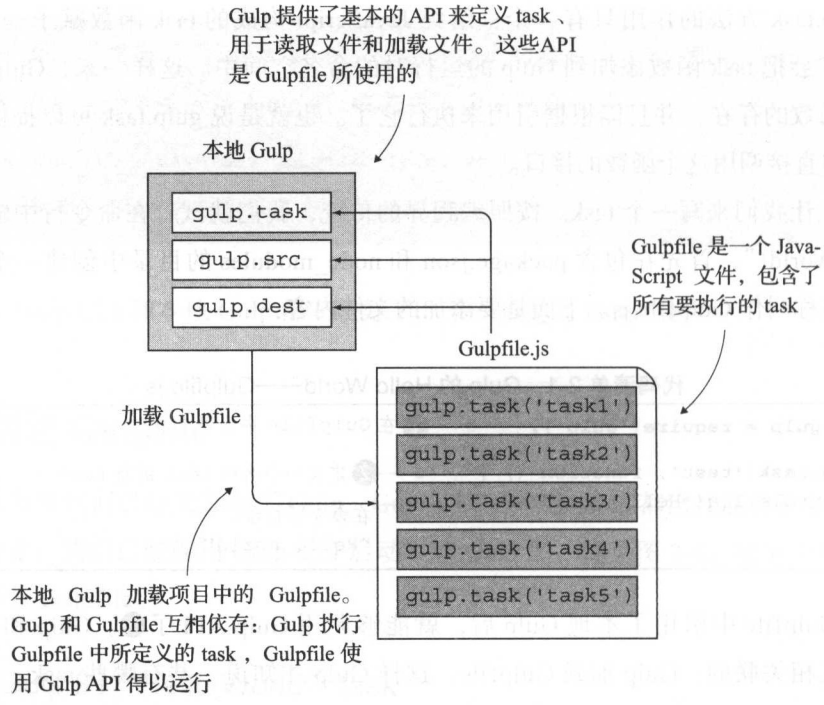



图 2-6 展示了本地 Gulp 和 Gulpfile 之间的相互作用。Gulp 加载项目中的 Gulpfile，并按照命令行的指令，执行其中的 task。Gulp 还为 Gulpfile 提供基本的 API，用来创建和定义 task

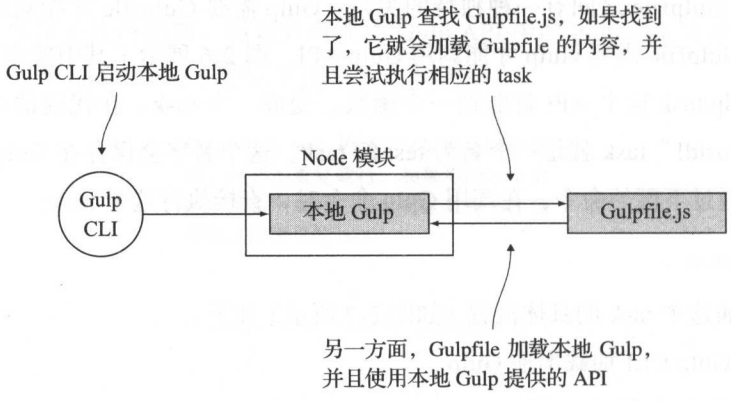



图 2-7 全局 CLI 启动本地 Gulp，本地 Gulp 查找 Gulpfile 并加载

输出的结果应该类似这样：

```
[15:01:06] Using gulpfile ~/Project/playground/test/gulpfile.js
[15:01:06] Starting 'test'...
Hello World
[15:01:06] Finished 'test' after 95 µs
```

现在你已经学会通过命令行执行已定义的 task 了。当然啦，一个“Hello World!”没什么用处。接下来就来做点实用的东西。

### 2.2.2 流的使用

通过 Gulp 你可以读取文件内容，然后把它转换成特定的形式，最后把众多 JavaScript 文件合并成一个文件。Gulp API 提供了一些方法来读写、转换文件，而在所有这些方法背后，其实都使用了流（stream）。

流在计算机领域并不是一个新概念，它起源于 1960 年代的早期的 Unix，流是从一个源（source）流向一个终点（destination）的数据的序列。源的形式有很多种类：文件、计算机内存、输入设备，比如键盘和鼠标。当一个流打开时，数据就会一块一块地从源头流出，然后被预先定义的过程处理。如果数据源来自于文件，那么每个字符或字节都会被依次读取；如果数据来源于键盘，那么每次按下的键都会通过流传输数据。相比于一次性加载所有数据，流最大的优点是：理论上，输入数据可以是无限的，而且没有任何限制。对于键盘而言，这一点就很有意义。哪有人会关闭控制电脑的输入流呢？输入流也被称之为可读流（readable stream），意思是从数据源读取数据。相对应的，还有还有向外输出的流。它可以是文件，也可以是内存里的数据，还可以是输出设备，比如命令行、打印机、显示器。这些流也被称之为可写流（writable stream），意思是它们将来自流的数据保存起来。图 2-8 展示了流的工作原理。

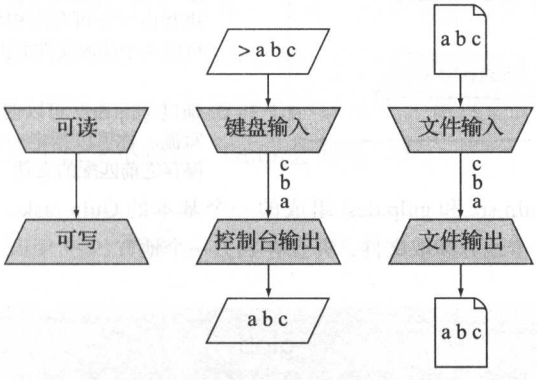



图 2-8 流可以分为两种：可读流和可写流。可读流接收数据，可写流写入数据

数据是一个元素的序列，随着时间流逝会不断产生新的元素（比如字符或字节）。可读流可以来自于不同的源，比如输入设备（键盘）、文件或内存中的数据。可写流也

可以结束于不同的地方，比如文件、内存、命令行。两种流可以交替：键盘输入可以结束于一个文件，文件输入可以结束于命令行。

流不仅可以有无限的输入，而且不同的可读流和可写流还可以结合起来。比如说，键盘输入的数据可以直接存储到一个文件中，又或者是把文件输入用命令行或打印机打印出来。而在这个过程中，接口始终保持不变，无论源头和终点是什么。

### 2.2.3 Gulp中的可读流和可写流

在Gulp中，你可以使用gulp.src方法来创建文件的可读流，它可以用来选择你想在task中处理的文件。不过由于你可能要处理很多不同的文件，而且很可能不知道文件具体叫什么名字，所以你可以用glob来定义匹配规则（selection pattern）。与可读流gulp.src相对的是可写流gulp.dest，它用来定义文件保存的位置。gulp.dest接受一个字符串参数，用来表示文件目录。这个目录是相对于Gulpfile所处的目录的。如果这个目录不存在，Gulp会创建一个。图2-9展示了上述流程。

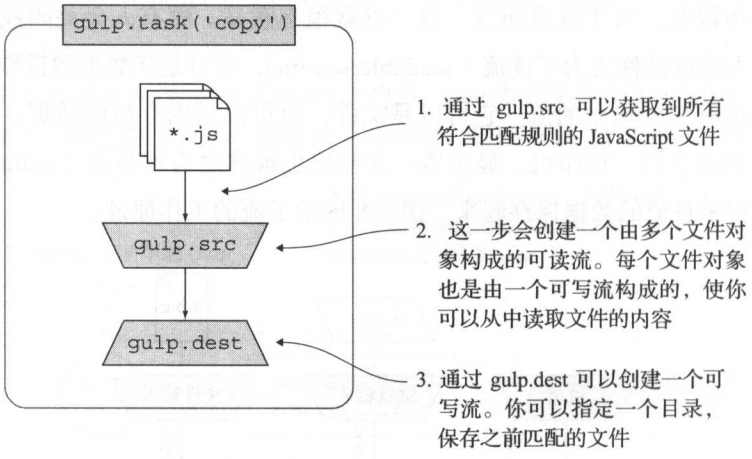



图2-9 由gulp.src和gulp.dest组成的一个基本的Gulp task。它的作用是从一个地方读取文件，并保存到另一个地方

#### glob

glob是计算机科学和编程领域的常见概念。如果你曾经用过命令行来删除所有某一特定类型的文件，那你应该很熟悉glob的工作原理。*.js中的星号是通配符，表示“任何名称”。如此一来，就能匹配到所有app/scripts目录下的JavaScript文

件。Node的glob更加高级，它支持更多复杂的规则。比如说有一个规则是在*.js*前面加两个星号，这个规则叫作globstar。它也是一种通配符，意思是匹配目录中的0个或多个子目录。比如说，app/script/*.js这个glob只有一个通配符，它能够匹配到scripts目录下的所有JavaScript文件，但是无法匹配到scripts的子目录下的文件。如果用了globstar规则，就能变成app/scripts/**/*.js，那就能匹配到子目录下的JavaScript文件了。

目前为止，我们已经知道了gulp.src可以根据特定规则来读取文件，gulp.dest可以把文件保存到特定目录中。明白这两点以后，就可以用这一部分Gulp API来创建第一个Gulp task了。既然gulp.src能读取文件，gulp.dest能写入文件，那么把这两者结合起来就能把文件复制到另一个地方。下面的代码就是这个task的实现。

**代码清单2-2 把文件复制到另一个目录——Gulpfile.js**

```
var gulp = require('gulp');

gulp.task('copy', function() {
  return gulp.src('app/scripts/**/*.js')
    .pipe(gulp.dest('dist'));
});
```

❶ 创建一个名为copy的新task

❷ 创建一个文件对象的可读流

❸ 流提供了pipe方法

首先创建copy task，然后所绑定的回调函数才能在Gulp的运行环境中执行❶。这里的glob规则匹配了app/scripts目录及其子目录下的所有后缀名是'js'的文件❷。然后，通过pipe方法把流的内容传入其他函数❸。在这个例子中，就是传入到了gulp.dest函数。这个函数把流中的文件保存到特定的目录中。在这个例子中，也就是保存到了dist目录下。如果这个目录不存在，Gulp会创建一个。



注意 从命令行运行gulp copy后就会开始运行执行链。在这个例子中也就是运行copy task。

gulp.src会打开一个可读文件流。所要处理的数据块都是根据glob规则匹配到的文件。一旦流打开后，就可以通过管道（pipe）把流导向特定的处理方法。管道本质上并不是Gulp的特性，而是Node流的一个概念。在代码清单2-2中，我们把流导向了gulp.dest方法。gulp.dest方法创建了一个可写流。可写流可以理解为一个数据池，是

数据流的终点，数据可以输出到屏幕或者文件中。在这个例子中，数据输出到了文件系统中。gulp.dest 创建的可写流所接受的数据类型和 gulp.src 创建的可读流所传出的数据类型是一样的。

现在你已经学会从文件系统中读取文件，并把它们保存到文件系统中，还学会了复制操作。但不得不承认，复制这个功能虽然重要，却没什么意思。接下来，让我们往流里面再加点料。

## 2.3 使用 Gulp 插件处理 task

到目前为止，你已经能够用 Gulp 在命令行中执行 task，并从文件系统中读取文件再复制到另一个地方了。但是只有当开始使用插件时，你才能领略到 Gulp 真正强大之处。Gulp 插件是一个小程序，用来在流中转换文件。这一节就将展示 Gulp 利用插件能做到哪些事情。

### 2.3.1 转换数据

流所擅长的可不仅仅是在不同的输入输出之间传递数据。当流打开时，开发者可以在数据传递到终点之前转换数据，比如把所有小写字母变成大写字母之类的。

这是流最强大的功能之一。当流打开时，你可以一块一块地读取数据，还可以在流中插入各种函数，图 2-10 展示了这个过程。

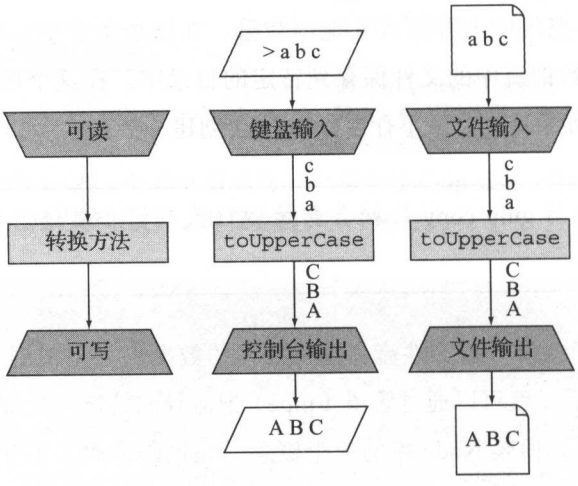



图 2-10 流不仅擅长传递数据，还擅长处理数据

要修改数据，就要在输入和输出之间添加转换方法（transformation）。在这个例子中，来自各个不同的源的数据被传入到了 toUpperCase 方法中。这个方法会把所有小写字母转换为大写字母。转换方法定义一次，就能在不同的流中复用。

在 Gulp 中，转换方法是通过插件来实现的。Gulp 的生态包含了超过 1500 多个插件，能让你用各种手段转换数据。下面让我们来看看如何通过 JavaScript 界最常见的转换步骤——Uglify（混淆），来转换 JavaScript 文件。

Uglify 是用 JavaScript 写的用来压缩文件的库。它能删除所有多余的空格，还能在不影响 API 的前提下尽可能缩短变量和函数名，甚至还采用了本书中介绍的各种优化 JavaScript 代码的方法（比如把 true 写成 !0，因为这样能节省两个字节）。这样一来，代码就会变得非常小，从而能够更快地被解析和传输。以流行的 jQuery 为例，经过压缩后，jQuery 的体积能从 250KB 缩小到 90KB，缩小了将近三分之一。不过一旦代码经过混淆后，人类就很难阅读了，这也是为什么它叫做 Uglify。Uglify 有一个 Gulp 插件，可以用下面的命令安装：

```
$ npm install --save-dev gulp-uglify
```

这行命令能够安装 Uglify 的 Gulp 插件，并且把对应的记录保存到 package.json 中。接下来就可以在 Gulpfile 中使用这个插件，具体如下所示。

**代码清单 2-3 混淆 JavaScript——Gulpfile.js**

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');

gulp.task('scripts', function() {
  return gulp.src('app/scripts/**/*.js')
    .pipe(uglify())
    .pipe(gulp.dest('dist'));
});
```

在引入 Gulp 之后，再引入之前安装好的 gulp-uglify 模块。这个模块可以把 Gulp 可读流中的内容加以混淆

在创建可读流之后，创建可写流之前，使用这个插件。用法非常简单，就像调用一个函数一样

每个通过流传递到 Uglify 的文件，都会转换成对应的混淆过的文件。如果通过 gulp script 命令来执行这个 task，然后再看看 dest 目录，你会发现原来一清二楚的 JavaScript 文件统统变成了一堆无法阅读的天书。任务完成！

## 2.3.2 改变文件结构

在加载 JavaScript 应用时，发的请求数量越少越好（原因参见第 1 章），所以我们要尽量把所有的 JavaScript 文件合并成一个文件。这个 task 通过 concatenation（合并）

完成，concatenation 能够把很多文件的内容合并到一个文件中。这个新文件拥有一个新的文件名和所有旧文件的内容。Gulp 和虚拟文件系统（virtual file system）也是如此。

在标准的流中，一般把文件视为数据的输入源，然后在之后的流中处理这些数据。当流打开时，所有文件的元信息，比如路径、文件名，都会丢失。但是因为要处理的文件往往不是一两个，而是一大堆，所以 Gulp 需要这些元信息才能处理文件。想象这样一个场景，你有 20 个 JavaScript 文件需要压缩。要处理这些文件，你必须要记住每个文件的文件名，并且记录哪些数据属于哪些文件，这样才能在最后输出数据时，还原出压缩的内容和原来的文件名之间的对应关系。

幸运的是，Gulp 能帮你解决这件事。Gulp 会同时创建一个输入源和一个数据类型，以供流来使用。这叫做虚拟文件对象（virtual file object）。它的结构有点特殊，如图 2-11 所示。

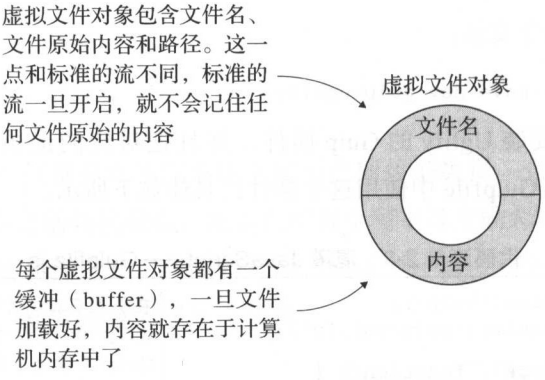



图 2-11 虚拟文件对象的示意图，虚拟文件由两部分构成：文件名和路径，以及一个包含文件原始内容的缓冲。这两部分信息都保存在计算机内存中

当 Gulp 流刚打开时，它会把所有的原始文件封装到一个虚拟文件对象中，之后在虚拟文件系统或者 Vinyl 中（这取决于 Gulp 调用了哪个软件）处理这些文件。

Vinyl 对象是虚拟文件系统中的文件对象，包含两个信息：文件的路径（在虚拟文件对象中是文件的名字）和文件的内容。这些虚拟文件存储在计算机的内存中，所以计算机能够以最快的速度处理它们。你所做的所有的文件修改，一般都是存储在硬盘上的，而 Gulp 把所有数据存储在内存里，没有在处理过程中使用昂贵的读写操作，所以能够飞速地处理数据。

你还可以使用虚拟文件系统在 Gulp task 中修改文件的结构。在合并的时候，你会

获得一个包含所有文件内容的虚拟文件，然后给它一个新的文件名就行了。详情请看下面的代码：

**代码清单 2-4 Gulpfile.js——合并文件**

```
var gulp = require('gulp');
var concat = require('gulp-concat');

gulp.task('scripts', function() {
  return gulp.src('app/scripts/**/*.js')
   .pipe(concat('bundle.js'))
      .pipe(gulp.dest('dist'));
});
```

❶ 引入一个名叫 gulp-concat 的包来处理合并过程

❷ 在创建可读流之后、创建可写流之前，使用该插件。

gulp-concat 模块可以预先通过 npm install --save-dev gulp-concat 来安装❶。第❷步把所有的 JavaScript 文件内容传入到 concat 步骤中。concat 插件需要一个参数：新文件的文件名。concat 插件会用这个新名字创建一个新的虚拟文件对象。这个虚拟文件对象的内容就是之前流中的所有的文件的内容。

### 2.3.3 链式调用插件

在前面的例子中，我们使用了不同的程序来转换输入的内容，再把它们保存到硬盘中，但 Gulp 所能做得远远不止这些。你可以在输出流之前，在流中插入任意数量的程序，从而让数据流经过多个转换步骤。

一个转换步骤应该只做一件事，并且把这件事做到最好。如此一来，我们就能通过组合多个转换步骤，创建一个更高级、更复杂的程序。在此引用一下 Doug McIlroy 的话，他就是在 1964 年为 Unix 操作系统提出流的概念的人：

我们应该要有多种方法来连接各个程序，就像连接水管一样。当需要用另一种方法来处理数据时，就切换到另一节水管。

Gulp 的精髓就是把数据流传入到一系列 task 中加以处理，参考图 2-12，其中展示了链式调用 uglify 和 concat 的过程。

现在你已经学会如何合并和压缩文件了。所有的过程都发生在计算机的内存里，完全不需要费时的硬盘读写操作。这使得 Gulp 即快速又灵活。按照下面的步骤，你就可以创建更复杂的程序了：

①创建一个 script task，用来合并和压缩文件。

②创建一个 style task，用来编译和压缩“less”文件，以及运行 Autoprefixer 来自动添加浏览器前缀。

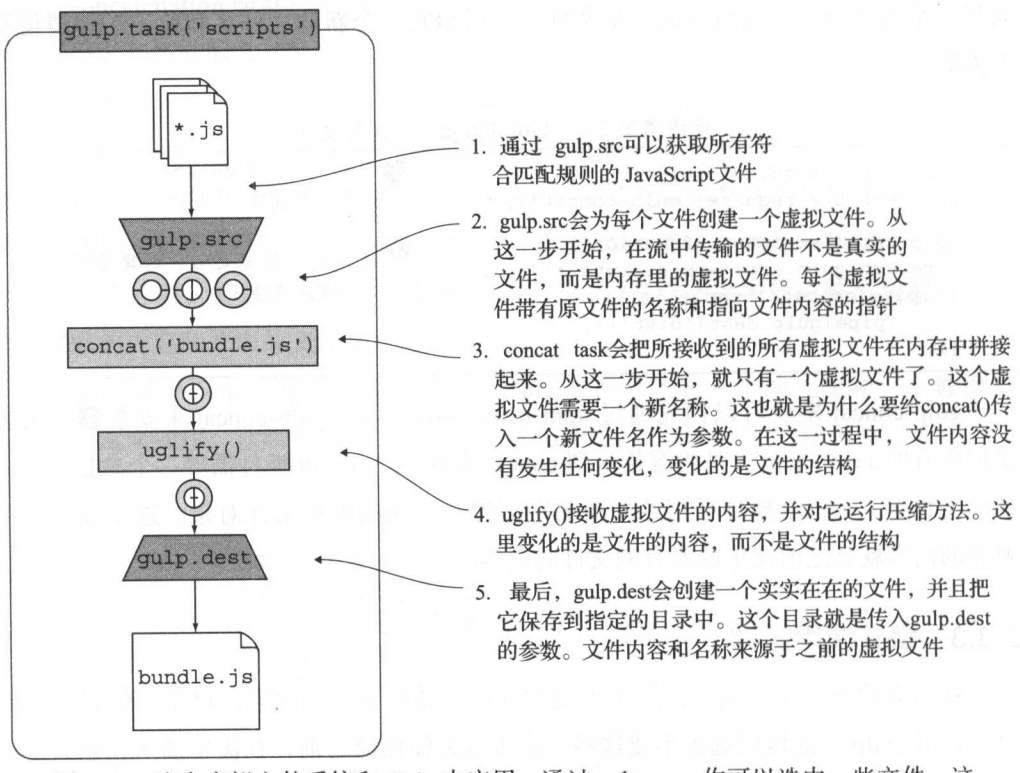



图 2-12 流在虚拟文件系统和 Gulp 中应用。通过 gulp.src，你可以选中一些文件，这些文件虚拟文件系统中成为虚拟文件对象。你可以访问虚拟文件对象的内容，然后利用 Gulp 插件实现一系列的转换步骤，以此来修改文件

③创建一个 test task，用来检查 JavaScript 代码质量，确保代码风格良好。

### 浏览器前缀

浏览器前缀是浏览器为实验中的特性添加的标记，这些特性在不同的浏览器中的语法和功能可能会有所不同。浏览器前缀是特殊的缩写，一般加在 JavaScript 方法或者 CSS 属性的前面，用以区分其他浏览器的实现方法、自己的实现方法和最后制定的标准。比如说，-webkit-animation 是 CSS 属性 animation 添加了前缀的版本，它专门用于 Webkit 渲染引擎。

下面的 Gulpfile 能够处理 JavaScript 和 CSS，还能够执行测试。还是像之前一样运行 npm install --save-dev <plugin-name> 来安装没有的插件。

**代码清单 2-5 完整的 Gulpfile.js**

```
var gulp        = require('gulp');
var jshint      = require('gulp-jshint');
var uglify      = require('gulp-uglify');
var concat      = require('gulp-concat');
var less        = require('gulp-less');
var minifyCSS   = require('gulp-cssnano');
var prefix      = require('gulp-autoprefixer');

gulp.task('scripts', function() {
  return gulp.src('app/scripts/**/*.js')
    .pipe(concat('main.min.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist/scripts'));
});

gulp.task('styles', function() {
  return gulp.src('app/styles/main.less')
    .pipe(less())
    .pipe(minifyCSS())
    .pipe(prefix())
    .pipe(gulp.dest('dist/styles'));
});
gulp.task('test', function() {
  return gulp.src(['app/scripts/**/*.js',
➥'!app/scripts/vendor/**/*.js'])#D
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
    .pipe(jshint.reporter('fail'));
});
```

❶ 引入这个 Gulpfile 中所有需要的模块。

❷ scripts task

❸ styles task

❹ test task

❺ JSHint 和之前的 task 有所有不同

用 npm install --save-dev <plugin-name> 安装每一个模块❶。script task ❷加载所有 app/scripts 目录下的 JavaScript 文件，并且加以压缩和合并。style task ❸加载一个 LESS 主文件，并且传入到三个处理方法中：用 LESS 编译、用 CSS Nano 压缩 CSS 和用 Autoprefixer 自动添加浏览器前缀。test task ❹和 script task 一样会加载所有脚本文件。但除此以外还有一个筛选规则，就是排除掉所有 vendor 目录里的文件。这是因为一般人们不会去检查 vendor 目录第三方库的代码风格。JSHint 的输出报告不会改变文件，只是告诉你哪些风格检查通过了而已❺。你可以把结果传入到 JSHint 的 reporter 插件中。

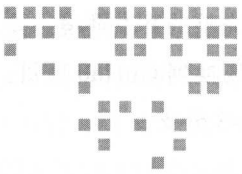
通过这短短几行代码，我们已经做了不少事。在命令行里运行 gulp scripts、gulp styles 或 gulp test，就能开始执行 task。现在，你已经有了一个 Gulpfile 来处理代码了。

## 2.4 总结

在这一章中，我们介绍了 Gulp 的入门知识：

- Gulp 的运行环境：Node.js 自带一个包管理器 NPM。NPM 可以全局安装 node.js 模块作为命令行工具，也可以在本地安装模块作为库来使用。Gulp 既有全局安装的命令行接口，也有在每个项目本地安装的库。
- Gulpfile 是用 JavaScript 编写的构建指令，它使用本地 Gulp 来调用 API。你可以在 Gulpfile 中引用 Node 模块，使用它们。
- Gulp 的 gulp.taskAPI 使函数能够在命令行中被调用。执行 gulp <taskname> 命令就可以运行在 Gulpfile 中定义的 task。
- gulp.src 和 gulp.dest 能够创建可读流和可写流，用来把文件从文件系统里的一个地方复制到另一个地方。
- 插件，比如 Concat 和 Uglify，可以用来转换 JavaScript 文件。
- 链式调用插件可以创造出更复杂的程序，比如一个既能压缩又能合并的 scripttask。又或者是一个能编译、压缩 CSS 文件，并且自动添加浏览器前缀的 style task。
- 使用 JSHint 进行代码风格检查可以确保代码质量。

通过这个简单 Gulpfile，你可以按需处理你的源代码。下一章我们会扩展这个 Gulpfile，使它能提供一个成熟的开发环境。



第 3 章 Chapter 3

# 用 Gulp 配置本地开发环境

**本章包括：**

- 依赖链。
- 顺序和并行执行链。
- 监控过程。
- 按需实时重载的本地服务器。

前面的章节讲了如何使用 Gulp 创建构建管道来链式执行多个转换方法。办法就是通过不同的规则匹配相应的文件，然后创建一个流输出这些文件，并通过若干专门的 Gulp 插件来修改文件内容。

虽然我们已经能创建不同的转换方法，而且只要按下按钮就能转换大量文件，但是距离大型的、完整的自动化还有一段距离。所谓的自动化所要求的就是要建立一个不需要开发者直接干预，便能自动执行大量任务的系统。换句话说，在整个开发周期中，开发者要做的就是把任务编写得井井有条，然后让自动化工具去做其余的工作。

Gulp 提供了自动化的能力。在本章中，你会学到其他 Gulp API 的用法。通过它们，你可以决定 task 执行的顺序。不仅如此，你还可以创建监控过程（watch process），它可以监控源代码文件的变化，如果有文件发生了改动，就会自动触发相应的 task。你甚至还能运行一个本地服务器，让它来自动刷新浏览器的页面，使 Gulp 成为一个完全

自动化的、完整的开发环境。要运行本章的示例，你可以在前面的章节的示例基础上修改，也可以切换到 GitHub 项目。（https://github.com/frontend-tooling/sample-project-gulp）中的 chapter-3 分支。

> **Gulp 4** 本章的大部分内容需要用到 Gulp 4 的特性，尤其是顺序和并行执行 task 的 API，是老版本 Gulp 所不支持的。我们强烈建议使用 Gulp 4 来运行本章示例。

## 3.1 本地开发环境

本章的主要目标就是创建一个本地开发环境，以满足快速开发迭代的需要。接下来，我们不会再手动地执行 task，而是定义一系列过程来让 Gulp 自动地在正确的时间，按照正确的顺序，执行正确的 task。

根据上一章里 Gulpfile 的任务，想象一下在平时工作中，你会怎么来操作（如图 3-1 所示）。你可能会按照下面的步骤来做：

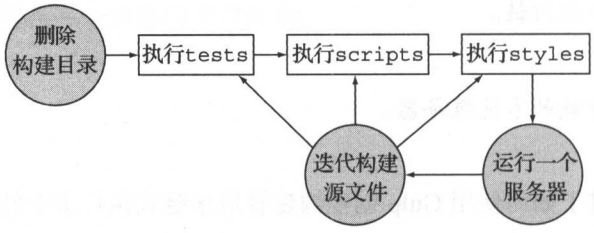



图 3-1 运行本地开发环境的步骤。首先清空构建目录，然后运行三个先前定义的构建 task，接着启动一个服务器。当服务器启动好以后，每当修改源代码后，就要再次运行 tests、scripts、styles 三个 task

①删除构建文件的目录（在上一章例子中叫作 dist），以保证在一个干净的环境下开始构建，免受上一次构建的影响。

②运行之前定义的 task：tests、scripts 和 styles。

③因为大部分 JavaScript 的功能都需要通过 HTTP 或者 HTTPS 传输文件才能实现，所以你需要运行开发服务器。

④开始迭代修改源文件，每当你添加了新功能或者新样式时，就要再次运行之前定义的 task。只要你还在开发，就要一直执行这个步骤。

前两步（删除构建目录和按顺序执行 task）会占用开发者不少精力。而且开发者常常忘记这两件事，要么忘记删除构建目录，要么搞错 task 的运行顺序，比如在运行测试前就先编译了代码。如果能够定义好 task 执行顺序并且让 task 自动执行，就能避免这些错误。第三步（启动开发服务器）很重要，而且能用多种的办法来实现。如果能够一次性定义好服务器的配置，之后很轻松就能运行的话，那就再好不过了。因为安装和配置服务器同样需要时间和精力，而其中大部分步骤又是相同的，所以这也是应该自动化的步骤。最后一步（迭代地构建源代码）同样需要不少精力，特别是在频繁地小范围修改代码的时候，一遍又一遍地重复这些 task 尤其枯燥。

本地开发环境能处理所有这些 task，它不仅知道要按什么顺序执行什么 task，还知道要把哪些文件放在开发服务器上。

### 3.1.1　给 task 分类

仔细观察图 3-1，你就会发现，所有任何都可以被分为三类，如图 3-2 所示：

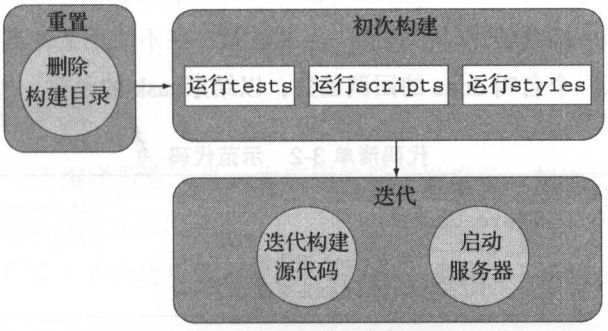



图 3-2　启动本地开发环境的前置步骤可以分为以下三步：创建一个干净的初始环境，初次构建源代码，迭代构建源代码的增量修改部分

①重置：删除上一次构建遗留的痕迹。

②初次构建：重新根据源代码编译文件。

③迭代：持续修改代码，并把结果展示在显示器上。

让我们仔细观察一下这些步骤。

### 3.1.2　重置步骤：clean task

重置步骤对于本地开发环境的作用非常简单：删除上一次构建遗留的痕迹。必须有一个干净的环境，才能确保之前的构建的残留不会干扰当前的开发环境。你需要创

建一个新的 task 来实现重置步骤。在第 2 章创建构建任务时，我们把构建的结果放在一个叫 dist 的目录里，现在我们要用 Gulp 删除这个目录中的所有东西。

Node.js 的生态系统有一个叫作 del 的包能做到这件事。你可以在上一章的 Gulpfile 里加入这个包，方法是在命令行里使用“npm install --save-dev del”命令来安装这个包，并且在 Gulpfile 里添加下面的代码。

**代码清单 3-1 Gulpfile.js——clean task**

```
var del = require('del');      <——在 Gulpfile 中引入 del 包

gulp.task('clean', function() {
    return del(['dist']);      <——调用之前引入的 del 包
});
```

把第一行和其他 require 声明放在一起。del 接受的第一个参数是一个数组，这个数组由要删除的文件的匹配规则构成（在这个例子中，就是 dist 目录）。del 本质上并不是一个 Gulp 插件，但是它的接口是兼容 Gulp 的。

正如上面的代码所示，用 Gulp 不一定非得用 Gulp 插件或者文件流，你完全可以把 Node.js 模块和 Gulp 结合使用。不过有一点要注意：在不使用文件流的情况下，需要向 task 的函数里传入一个名叫 done 的回调函数，以结束 task 的函数，如下面的代码所示。

**代码清单 3-2 示范代码**

```
gulp.task('report', function(done) {          <——让 Gulp 给 task 传入 done 回调函数
   console.log('We are done!');
   done();                                    <——调用 done 回调函数以通知 Gulp task 完成
});
```

done 回调函数的作用是在 task 完成时通知 Gulp（而不是返回一个流），而 task 里所有其他的功能都纯粹依赖 Node 的来实现。



**注意** 有一种 Gulp 的最佳实践，能使用 Node.js 模块，就使用 Node.js 模块。只有当要转换文件，并且要把文件保存到文件系统的某个目录时，才使用 Gulp 插件。Gulp 流有开始和结束，开始和结束都是事先定义好的，它们的作用把是文件系统中的原始文件和结果文件连接起来。尽管删除文件也是一种对文件的转换，但是删除文件并不会在文件系统中产生输出，也没有结果需要保存。

通过 clean task，可以完成重置步骤。接下来，让我们解决初次构建和迭代的问题。

### 3.1.3　构建和迭代

初次构建和迭代乍一看非常类似，因为它们执行的都是在第 2 章中定义好的 task，但是它们之间又有着微妙而重要的差异。

初次构建会按顺序运行所有编译 task（styles、tests、scripts），这种顺序不仅对开发者而言容易理解，而且能够保证构建结果之间不会互相干扰。不过按照我们目前的 Gulp 配置，虽然在构建时不会产生干扰，但是在构建的开始和结束时却有可能产生。所以构建目录 dist 必须要在编译开始之前清理，否则就有可能删除新编译的文件。同样的，服务器要在编译完成之后启动，否则服务器启动时，文件可能还没有编译好，这都是初次构建阶段需要考虑的事情。

另一方面，迭代阶段只会根据源代码的增量改动部分执行所有构建 task 中的一部分 task。当一个 JavaScript 文件被修改时，并不需要运行 styles task，因为两者之间没有关系。反之亦然，修改 LESS 文件也不需要触发 scripts task。

这两个步骤比重置步骤更需要注意，因为它们需要在第 2 章的 Gulpfile 的基础上进行大改动。在接下来的两节中，我们会分别加以说明。

## 3.2　初次构建步骤

初次构建步骤负责编译所有文件，为应用的部署做准备。初次构建完之后，要么是直接部署应用，要么是继续迭代开发。

在此我们简要概括一下构建步骤需要执行的 task，如图 3-3 所示。

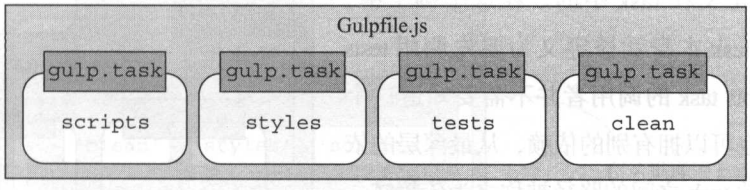



图 3-3　需要按顺序执行的四个 task：scripts task、styles task、tests task 和新增的 clean task。clean task 会删除构建目录，然后重新创建一个

步骤如下：

①删除之前的构建步骤遗留的构建目录，从一个干净的状态开始构建，这一步叫做 clean。

②运行 test task 测试 JavaScript 代码。

*③运行 styles 和 scripts 编译代码，并且把代码保存到正确的目录里。*

④为了按照顺序执行这些 task，你需要定义不同的依赖链和执行链，并且通过 gulp.series 和 gulp.parallel 把它们串联起来。gulp.series 用来按顺序执行 task，gulp.parallel 用来并行执行 task。

### 3.2.1 依赖链和执行链

当你仔细研究每个 task 的目标和 task 之间的关系的时候，你会发现一个细微却重要的不同之处。如图 3-4 所示，每次运行 scripts task 前都要运行 test task，也就是说 scripts task 依赖 tests task 的输出结果。styles task 则不同，它没有这层依赖关系。但是在运行 scripts 和 styles task 之前，都需要清空 build 目录。所以我们不得不定义一个执行顺序：首先，执行 clean task，再并行执行 styles 和 scripts task，但是后者要等 tests task 执行完后才能开始执行。

#### 1. 依赖链

这里首先要引入的概念是依赖链（dependency chain），在图 3-4 中表现为虚线箭头。在这个例子中，scripts task 和 tests task 是紧密耦合的。谁也不想不经过测试就直接运行 scripts task 编译源代码。所以每当要执行 scripts 时，必须要先执行 tests task。我们把这样的关系叫作依赖。

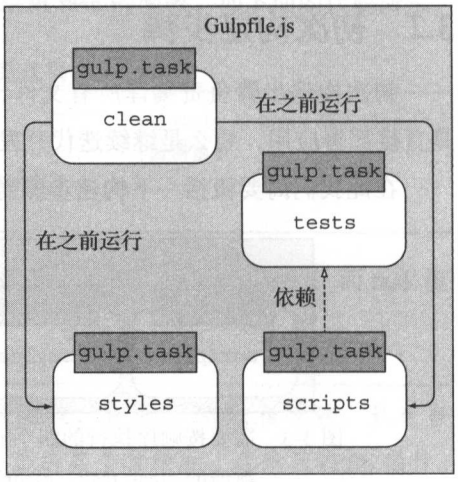



图 3-4 task 之间的关系

依赖是定义在 task 中的。在这个例子中，scripts 这个 task 本身就被定义为要先调用 tests task。而 scripts task 的调用者并不需要知道这个细节。依赖也可以拥有别的依赖，从最深层的依赖到最顶层的 task 之间的路径被称之为依赖链。

#### 2. 执行链

第二个概念是执行链（excecution chain），在图 3-4 中表现为实线箭头。有些 task 需要按照特定顺序执行，比如执行 styles 和 scripts task 之前要执行 clean task，但是它们之间又没有耦合到依赖的程度。因为即使不先删除构建目录，直接执行 styles 和 scripts task，也没什么关系。只有在特殊情况，

比如初次构建时，才需要调用 clean task。为了描述这个关系，我们需要定义一个概念叫作执行链，意思是一系列按顺序调用的 task，可以是顺序调用，也可以是并行调用。

Gulp 提供了两个函数来实现依赖链和执行链。

### 3.2.2　顺序和并行执行 task

你可以使用两个 Gulp API 中的 task 执行函数：gulp.series 和 gulp.parallel，来创建执行链。顾名思义，它们分别是用来顺序和并行执行 task 的：

- gulp.series：这个函数用来按顺序执行 Gulp task。它可以接受无限个参数，这些参数既可以是字符串（代表一个已定义的 Gulp task），也可以是函数。
- gulp.parallel ：这个函数接受的参数和 gulp.series 一样，区别在于它是并行执行 task 的。为了实现这一点，它会产生多个进程来同时执行这些 task。

图 3-5 展示了如何用 gulp.series 定义一个顺序的执行链。

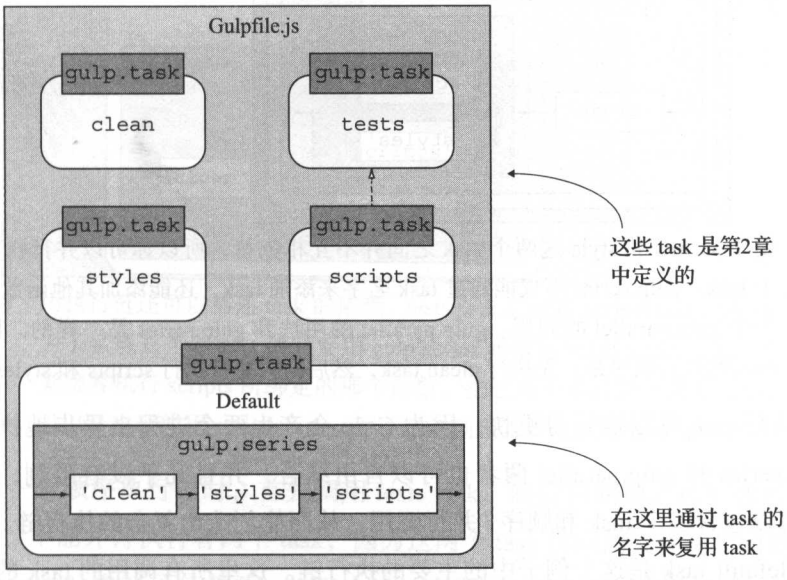



图 3-5　一条顺序的 task 执行链。通过 gulp.series 函数，你可以按顺序执行 clean、styles 和 scripts task。这个 API 接受 task 的名字作为参数

可以看到，当你创建一个新 task（名叫 default）时，本质上其实这个 task 并没有对应的函数，只不过是调用了 gulp.series，执行了定义好的三个 task。这三个 task 会按照被传入到函数里的顺序执行，首先运行 clean task，然后是 styles，最后是 scripts。注

意，我们暂时省略了 tests task。gulp.series 可以保证这几个 task 严格按照顺序依次执行。

styles 和 scripts 并没有共同的依赖，因此可以并行地执行它们。使用 gulp.parallel 函数，就可以这两个 task 结合起来，让他们在整体的流程中表现得就像是一个 task，但是彼此之间却又是并行执行的。细节参见图 3-6。

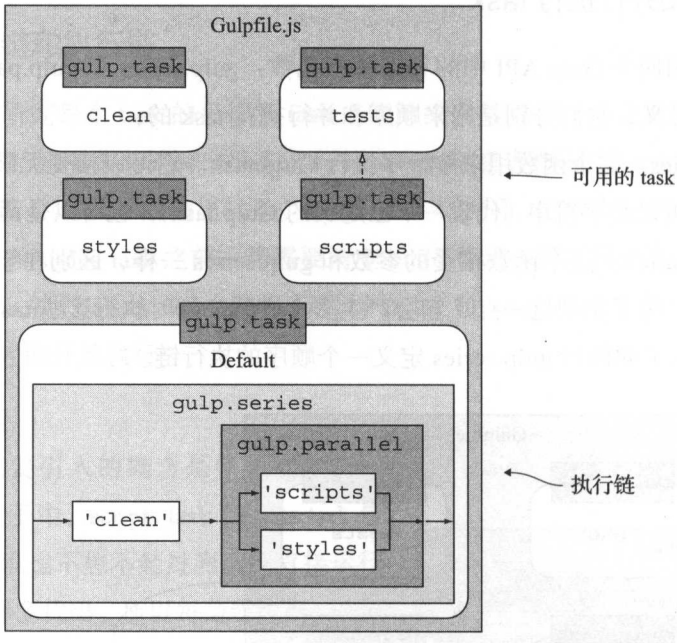



图 3-6 由于 scripts 和 style 这两个 task 之间并不互相依赖，所以你可以并行执行这两个 task。gulp.series 不仅能通过 task 名字来添加 task，还能添加其他函数，比如一个 gulp.parallel 的调用。gulp.parallel 的用法和 gulp.series 是一样的。图中的 task 的执行顺序是，先执行 clean task，然后同时并行执行 scripts 和 styles task

并行执行 task 可以执行得更快，因为 Gulp 会产生两个进程来异步地执行这两个 task。gulp.series 和 gulp.parallel 两者也可以自由结合，并且几乎没有限制，因此这两个方法都可以包含多个 task 和顺序 / 并行调用，从而能创造出复杂的执行链。

新的 default task 是这个例子中的主要的执行链。这里所有调用的 task 的目的都是为了构建和启动项目。

### 3.2.3 依赖链的 task 执行函数

gulp.series 和 gulp.parallel 还可以用来创建依赖链。创建依赖连和创建执行链的区别在于在什么地方调用。还记得之前省略的 tests task 吗？之前之所以省略它，是因为

我们要把它和 scripts 紧密耦合在一起，以此确保每次调用 scripts task 时都会事先调用 tests task。图 3-7 展现了怎么修改 scripts task 来让它和 tests 紧密耦合。

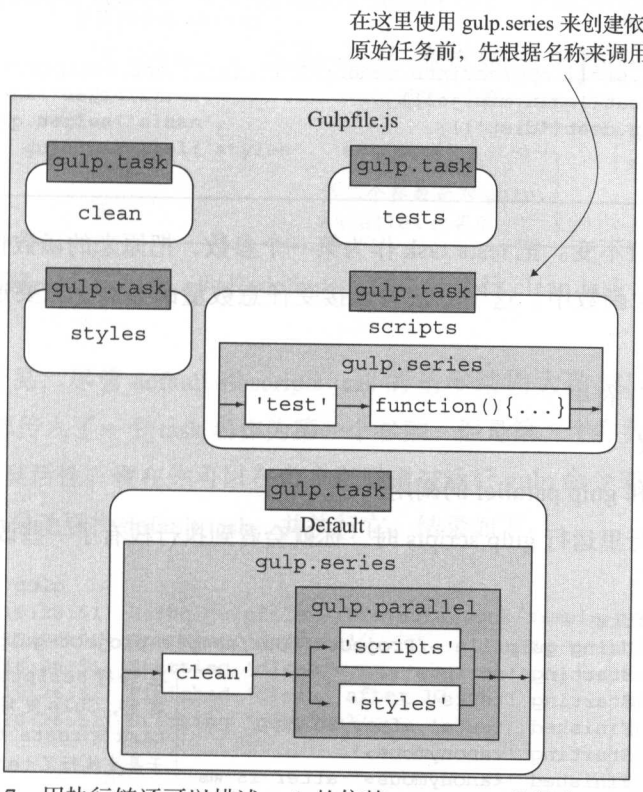



图 3-7　用执行链还可以描述 task 的依赖，tests task 必须在 scripts task 之前运行。为了实现这个目标，我们要用 gulp.series 函数来先执行 tests task，然后紧接着执行 scripts 所绑定的那个函数。把这两个函数紧密耦合之后，就可以自由地使用 scripts task，而不用时刻想着要先执行 tests task 了

创建另一个顺序执行链时，首先调用 tests task，然后再调用之前创建的 task。在这个例子中，不能并行执行者两个 task，因为这两个 task 访问的文件是相同的。

给 scripts task 添加了依赖后，default task 的调用就变得有点复杂了。原来调用 scripts task 的地方，现在替换成了 gulp.series 产生的 task 链，并且同时还嵌套在并行的 task 链中，让我们来把这些改动呈现到 Gulpfile 中。

### 3.2.4　更新 Gulpfile

首先，你要在 scripts 函数中调用依赖，如下面的代码所示。

**代码清单 3-3 Gulpfile.js——修改 scripts task**

```
gulp.task('scripts',
    gulp.series('tests', function() {
      return gulp.src(['app/scripts/vendor/**/*.js', 'app/scripts/**/*.js'])
        .pipe(concat('main.min.js'))
        .pipe(gulp.dest('dist'));
    })
);
```

第一个参数保持原样

第二个参数现在是一个 gulp.series 函数的调用

task 名 scripts 保持不变。把 tests task 作为第一个参数，把原来的函数作为第二个参数，传入 gulp.series 函数中。这个方法可以接受任意数量的参数，只要这些参数符合以下任意一个条件：

- 代表 task 名的字符串
- JavaScript 函数
- 对 gulp.series 和 gulp.parallel 的调用

现在当你在命令行里运行 gulp scripts 时，你就会看到执行栈有了一些改进：

```
$ gulp scripts
[18:09:26] Using gulpfile ~/Projects/book/sample-project-gulp/gulpfile.js
[18:09:26] Starting 'scripts'...
[18:09:26] Starting 'tests'...
[18:09:26] Finished 'tests' after 30 ms
[18:09:26] Starting '<anonymous>'...
[18:09:26] Finished '<anonymous>' after 15 ms
[18:09:26] Finished 'scripts' after 48 ms
```

调用了 scripts task

在运行 scripts task 的函数前，Gulp 发现了 scripts task 对 tests task 的依赖，于是先执行了 tests task

在 tests task 之后，执行了 scripts task 绑定的匿名函数

完整的 scripts task 执行完毕，这里给出的时间，是整个 scripts task 花费的时间，包括了 tests task 和和匿名函数花费的时间

之所以会产生调用 anonymous 的情况，是因为你传入的函数没有名字。它已经不是 scripts task 了，因为 scripts 现在是包含两个函数的顺序执行链。你可以通过给函数命名来修改它的名字，如下面的代码所示。

**代码清单 3-4 Gulpfile.js——给匿名函数命名**

```
gulp.task('scripts',
    gulp.series('test', function scriptsInternal() {
        … // same as before
    })
);
```

你可以随便给这个函数起个名字

这个函数无法在全局作用域中被调用，它只会被 Gulp 打印出来。一个良好的命名规范是在原 task 的后面加上“Internal”，并且采用驼峰式命名。

修改完 scripts task 后，我们再来修改 default task。下面的代码就是图 3-6 的实现。

**代码清单 3-5　Gulpfile.js——使用 series 和 parallel**

```
gulp.task('default',                          <-- 创建一个新 task
    gulp.series('clean',                                  <-- 第一个参数是对 gulp.series 的调用
        gulp.parallel('styles', 'scripts')    <-- 第二个参数是对 gulp.parallel 的调用
    )
);
```

我们创建一个新 task，叫作 default。它先是执行 clean task，然后并行调用 styles 和 scripts。

如你所见，尽管 default 和 scripts task 的基本结构大致相同，它们在调用 gulp.series 时，都传入了一个 task 名作为第一个参数，但是第二个参数就不同了，这正体现了执行链的灵活性。现在你可以在命令行里通过运行 gulp 命令来调用新的 default task 了，Gulp 会自动寻找 default task，并执行它。结果如下：

```
$ gulp
[18:26:51] Using gulpfile ~/Projects/book/sample-project-gulp/gulpfile.js
[18:26:51] Starting 'default'...
[18:26:51] Starting 'clean'...
[18:26:51] Finished 'clean' after 12 ms
[18:26:51] Starting 'parallel'...
[18:26:51] Starting 'styles'...
[18:26:51] Starting 'scripts'...
[18:26:51] Starting 'test'...
[18:26:51] Finished 'test' after 164 ms
[18:26:51] Starting 'scriptsInternal'...
[18:26:51] Finished 'styles' after 185 ms
[18:26:51] Finished 'scriptsInternal' after 14 ms
[18:26:51] Finished 'scripts' after 189 ms
[18:26:51] Finished 'parallel' after 190 ms
[18:26:51] Finished 'default' after 206 ms
```

并行执行 scripts 和 styles

clean task 是第一个通过 default task 执行的 task，和 Gulpfile 里设置的一模一样。由于 defualt task 是顺序执行的，所以在 clean 完成之后，别的 task 才开始运行

scripts 和 styles task 同时开始，期间没有别的 task 开始或完成

scripts task 的执行顺序就和之前的示例一样

这里有一个值得注意的地方，styles task 完成得比 scripts task 早很多，Gulp 要等到这两个 task 都完成……

……才算完成这个并行执行链，从而进一步完成整个 default task

当你得到上面的结果时，就说明初次构建步骤已经完成了。接下来，让我们开始迭代步骤。

## 3.3 迭代步骤

完成初次构建后，就要在开发时不断地迭代地构建源代码。从现在开始，我们并不想每次都重新构建所有的代码，而是想只重新构建改变的增量部分。比如说，修改一个样式文件并不需要重新构建所有的 JavaScript 文件，因为 JavaScript 文件没有任何改变。

利用 Gulp 可以监控源代码的改动，并且每当发生发动时自动执行 task。这就是*监控过程*。为了能够直接在浏览器中看到修改的结果，你还需要启动一个本地服务器，并且把修改的结果注入到浏览器窗口中。

### 3.3.1 监控过程

文件监控过程会持续地轮训文件系统，以查看硬盘中是否有改动。由于 Gulp 的本质就是把文件从一个状态转换到另一个状态，所以 Gulp API 中包含监控功能也很合理。

通过 gulp.watch 函数，Gulp 提供了一个启动监控过程的接口，用来监控符合某一特定规则的文件的变化，并触发相应的 task。watch 函数会启动一个监控过程，以监控文件的变化。一旦文件发生了变化，就会触发回调函数，大部分时候是一组 Gulp task，但也可以是别的函数。图 3-8 展示了如何在 Gulpfile 中，向已有的 defualt task 添加文件 watcher。

在之前定义的顺序执行链中添加一个新函数，然后把这个新函数放到 scripts 和 styles task 之后，以确保它们和监控过程的 task 不会同时执行。因为 gulp.series 不仅能接受 task 名，还能接受函数作为参数，所以你可以在 gulp.series 之后再加一个函数作为参数。这个函数的内容是两个 watcher——或者是两个 gulp.watch 的调用。其中一个监控样式文件的变动，另一个监控 JavaScript 文件的变动。添加了 watcher 之后，Gulp 就会开始监控文件系统，一旦发生下面三种变动之一，就会触发 gulp.watch 所引用的 task：

- 现有文件的变动：每当有文件被保存到文件系统并且时间戳改变时，就会触发 task。
- 文件系统中新增了符合规则的文件：当新增文件时，会触发 task。
- 文件系统中删除了符合规则的文件：当删除文件时，会触发 task。

Gulp watcher 也是根据一个特定的 glob 规则来监控文件的，就像用 gulp.src 匹配文件时用的规则一样（比如，scripts/**/*.js 匹配的是 scripts 目录和子目录下的所有 JavaScript 文件）。每当文件系统发生变化时，watcher 就会注意到变化，然后检查这个变化是否和规则相匹配。如果匹配，watcher 就会调用作为第二个参数传入的函数。切记：这个函数可以是任何函数，gulp.series 和 gulp.parallel 的调用也可以。

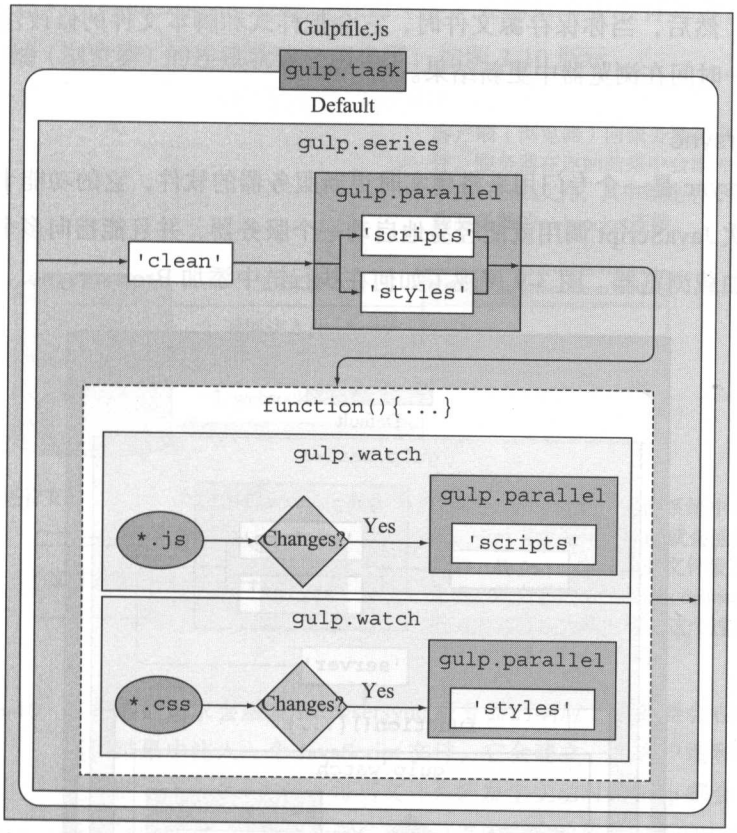



图 3-8　在执行链中的 clean、scripts、styles task 后，添加一个新的函数。在这个函数中，我们定义了两个 watch 任务。其中一个监控 JavaScript 文件，另一个监控 CSS 文件。当文件发生变化时，就会触发相应的 task。当 gulp.watch 函数被调用后，Gulp 进程就不会停止，会一直监控你所选择的文件。你可以通过 Ctrl-C 停止 Gulp

完成了监控步骤后，现在就可以增量构建迭代开发的源文件了。

### 3.3.2　使用 Browsersync 建立按需实时重载的服务器

对于迭代开发而言，运行一个本地开发服务器是很有必要的。因为它能提供一个非常棒的功能：在浏览器窗口里直接改变应用的修改结果，让浏览器重新加载当前页面和资源，让整个流程更加自动化。

通常，一台 PC 只能同时运行一个 Web 服务器，开发者要添加更多项目就要添加更多子域名和目录。这是一项费时的工作，但很容易自动化。借助 Browsersync，就能做到这一点。首先，启动一个新的开发服务器，专门托管你的项目，这个服务器随时

都可以关闭。然后，当你保存源文件时，它会把样式和脚本文件的修改注入到浏览器中，并且第一时间在浏览器中更新结果。

### Browsersync

Browsersync 是一个专门用来搭建实时更新服务器的软件，它的功能十分强大。它只需要用一次 JavaScript 调用就能轻易地启动一个服务器，并且能指向多个目录。它还能按需重新加载浏览器。图 3-9 展现了如何在执行链中添加 Browsersync。

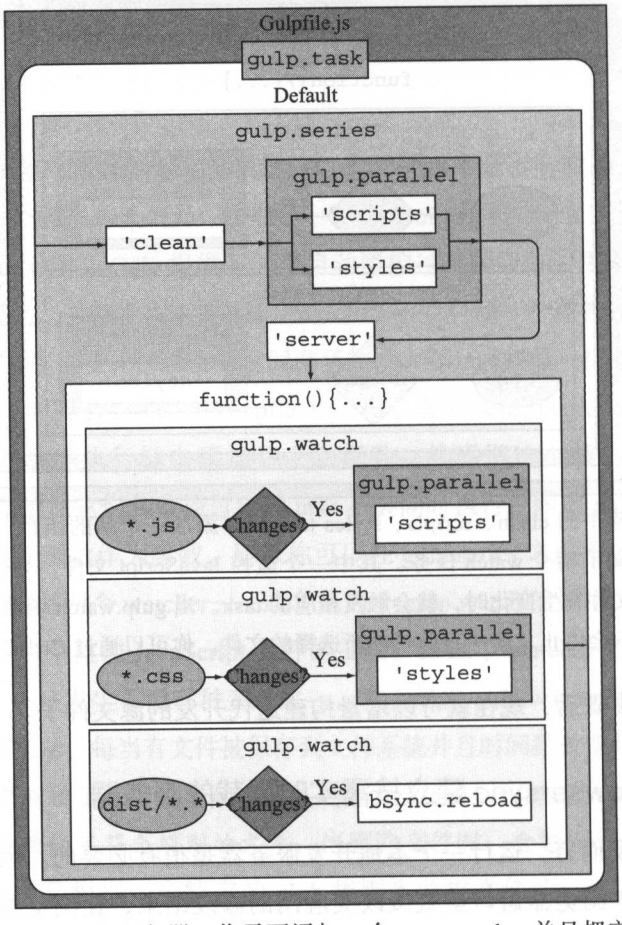



图 3-9 为了运行开发服务器，你需要添加一个 server task，并且把它放到执行链中的正确的位置。由于服务器需要在 watcher 启动之前、构建结束之后运行，所以要把 server task 放在 series 中的 parallel 之后。同时你还需要添加一个 watcher，这个 watcher 监控 dist 目录中所有文件的变化，但是在回调中，调用的并不是 Gulp 的函数，而是调用 Browsersync 的 reload 方法

当服务器和文件的 watcher 都建立好时（接下来的章节中会详细介绍其实现），服务器和客户端（浏览器）的连接就建立起来了，如图 3-10 所示。

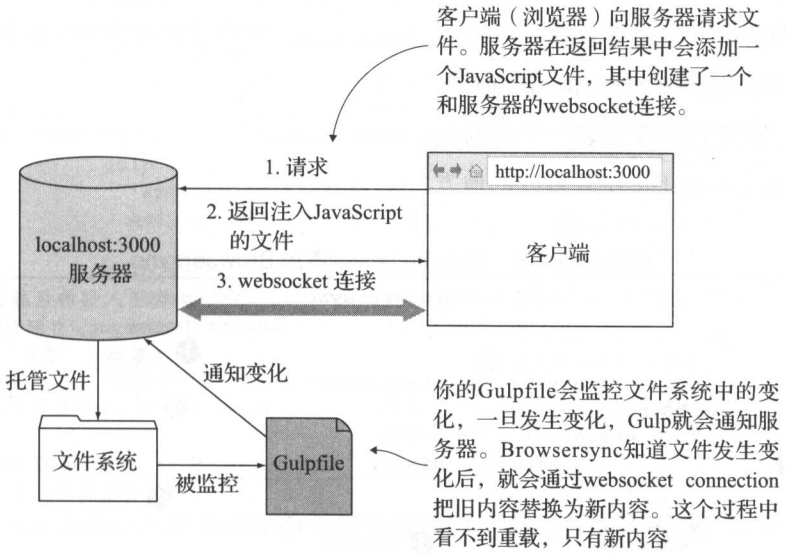



图 3-10　客户端的请求会触发 Browsersync 服务器的响应。服务器会在响应的返回结果中注入一个 JavaScript 文件，它会建立一个客户端和服务器之间的 websocket 连接。每当文件系统发生变动时，Gulp 都会通知服务器这个变动，然后服务器会把浏览器中内容的替换为新的内容

每当 Web 服务器收到一个 HTML 请求时，Browsersync 都会在返回结果的最后注入一小段 JavaScript 代码，这段 JavaScript 代码会建立一个和服务器的 websocket 连接。Websocket 是一种让客户端和服务器直接通信的方法，不需要用户操作或者 HTTP 请求。这样一来服务器就能实时地通知客户端的 Web 应用了。Websocket 一般被用来更新少量数据。在我们这个场景中，这些数据就是 JavaScript 和 CSS 中新增的内容。Browsersync 不仅仅会传输文件内容，还会传输不同客户端中的滚动事件和点击事件。也就是说，你可以打开多个浏览器访问同一个网站，Browsersync 会确保每个浏览器展示的内容都是一样的，这也是为什么它的名字叫做 Browsersync。

### 3.3.3　更新 Gulpfile

在这一节里，你会学到下面这些概念，从而进一步地完善自动化。

- 监控过程可以监听文件改动，并且触发一个或多个 task。
- 一个客户端和服务器之间的 socket 连接可以用来注入样式和脚本等资源。

这两个概念已经在图 3-9 中结合到了我们的构建管道中，接下来就要对 Gulpfile 进行相应的修改。我们要创建一个新的 server task，因为它既可以集成到构建执行链中，也可以被当作一个独立的组件使用。

首先，用这行命令安装一个新的包：`npm install --save-dev browser-sync`。下面是 Gulpfile 中要修改的地方。

**代码清单 3-6 Gulpfile.js——添加 Browsersync**

```
var bSync  = require('browser-sync');          ← ❶ 像引入别的包那样引入 Browsersync，并把它保存在 bSync 变量中
...
gulp.task('server', function(done) {           ← 创建一个名叫 server 的 task ❷
    bSync({                                    ← ❸ 传入一个配置对象
        server: {
            baseDir: ['dist', 'app']           ← ❹ 给这个配置对象添加两个目录
        }
    });
    done();                                    ← ❺ 调用回调
});
```

把第❶行代码和 Gulpfile 中的其他 require 声明放在一起。你所传入的函数需要接受 done 回调函数作为参数，并且在之后调用 done❷。用简单的一行命令启动 Browsersync 服务器❸。配置对象❹定义了服务器的根目录，分别是 dist 和 app。当收到请求时，Browsersync 首先会在 dist 目录中查找文件。如果没找到，就会继续在 app 目录中找。如果两个目录里都找不到，Browsersync 就会产生一个 404 错误。服务器启动后，调用回调函数❺。

Browsersync 不需要很多参数，它的默认参数就足够运行一个服务器了。如果你的系统配置正确，Browsersync 还会帮你打开浏览器，新建一个标签，访问正确的 URL。如果你觉得这些默认参数满足不了你的话，你可以访问 http://browsersync.io，看看有哪些选项你可以修改。

有一点很重要，就是要调用 done 回调函数。因为 Browsersync 会启动一个新进程，所以如果不执行这个回调，Gulp 就会卡住，无法调用其他函数。done 回调函数能够确保你在上一个执行链中跳转到了下一步，如下面的代码所示：

**代码清单 3-7　Gulpfile——watcher 函数**

```
gulp.task('default',
  gulp.series('clean',
    gulp.parallel('styles', 'scripts'),
    'server',
    function watcher(done) {
      gulp.watch(
        ['app/scripts/**/*.js', '!app/scripts/vendor/**/*.js'],
        gulp.parallel('scripts')
      );
      gulp.watch(
        'app/styles/**/*.less',
        gulp.parallel('styles')
      );
      gulp.watch(
        'dist/**/*',
        bSync.reload
      );
      done();
    }
  )
);
```

首先，在执行链中添加 server task

❶ 添加一个 watcher 函数

通过 gulp.watch 调用第一个参数

传入一个数组作为第一个参数，这个数组由 glob 规则构成 ❷

不一定非要用数组，❸ 也可以用字符串

调用最后的 watcher ❹

❺ 传入 Browsersync 的 reload 方法作为第二个参数

最后一步，通知 Gulp watcher 函数已经结束了

给 watcher 函数命名就可以在命令行中看到对这个函数的调用❶。这里❷的 glob 规则和第 2 章中的是一样的。我们只要重新构建自己的文件就行了，而不用重新构建第三方的文件。然后并行地调用 scripts task。❸的 glob 规则指向 LESS 文件。最后一个 watcher 监视的是在 dist 目录里创建的所有文件。我们直接把 Browsersync 的 reload 函数作为第二个参数传入，而不是像之间一样传入一个 Gulp 函数。每当 dist 目录里的文件发生变化时，都会执行一次 reload 函数。

如果你现在执行 Gulp（在命令行里输入 gulp），你会看到整个执行链都开始运行了。首先是 clean，然后是 scirpts 和 styles 在并行执行（test task 在 scriptsInternal 之前执行）。当这两个 task 执行完毕后，就会启动服务器，并且运行文件 watcher。注意，Gulp 并不会停止，因为还有很多其他进程正在运行，它们不应该自己停止。如果要停止 Gulp 的话，只要在命令行里按下 Ctrl-C，所有的子进程也会随之停止。

到此为止，你已经实现了所有过程。本地开发环境已经完成了。

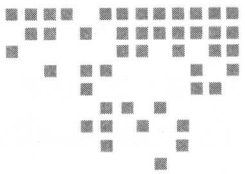
## 3.4　总结

本章主要说明了如何创建一个本地开发环境，大体流程是：重置、初次构建、迭

代。每一步都会用到第 2 章中创建的 task。你只要在一开始用一行命令，就能自动地执行这些 task 了。

- 为了初始化开发环境，我们用 Node.js 的 del 模块创建了一个 clean task，从而获得一个干净的开发环境。
- 初次构建是由执行链完成的。执行链是通过 Gulp API 中的 gulp.series 和 gulp.parallel 实现的，从而能顺序或者并行地执行 task。
- 本章还介绍了依赖链的概念。依赖链也是通过 gulp.series 实现的。依赖链让依赖和 task 紧密地耦合在一起，从而确保在编译 JavaScript 文件前一定会先测试 JavaScript 代码。
- 我们通过 Gulp 的文件 watcher 实现了增量迭代构建。watcher 会监视文件系统的变化，并且触发相应的 task。
- 我们还用 Browsersync 启动了一个本地开发服务器，这样就能看到每次修改的结果。watcher 会通知服务器注入文件的修改，并在浏览器中呈现出结果。

现在我们已经搭建好了一个本地开发环境，接下来我们要离开 Gulp 的世界，进入到工具链的下一环。因为目前我们的代码和第三方代码是放在一起的，所以我们要在构建时把第三方代码排除在外。但是我们不想这么麻烦，我们希望有一个地方专门放第三方代码，并且单独处理它们。



第 4 章 Chapter 4

# Bower 依赖管理

**本章包括：**

- 依赖管理简介。
- Bower 前端依赖管理工具。
- 扁平和嵌套的依赖树。
- 集成 Bower 的 Gulp 插件。

我们已经为 Gulp 创建了一个完善的构建配置文件，接下来就要进入到前端开发工作流的下一步：依赖构建工具。

所谓依赖，是指独立的、可复用的代码包。用依赖的时候不需要修改自己的代码，只管用就是了。依赖既包括像 Bootstrap、jQuery 这样的大型框架，也包括很多应用中的小组件，比如轮播图之类的。组件自己可能也有依赖，这些依赖有可能就是你已经在用的某个库。

你的代码越多，就越难管理。想象一下，你的网站上有 10 个 jQuery 插件，而你现在想要把 jQuery 升级到新版本。这个过程中，很难确保这些插件能够兼容新版本的 jQuery。

这个时候就需要依赖管理工具出厂了。像 Bower 这样的工具可以帮你你安装、搜索和更新依赖，如图 4-1 所示。它还能帮你解决依赖间的交叉引用，以及现有框架和

组件之间潜在的版本冲突。

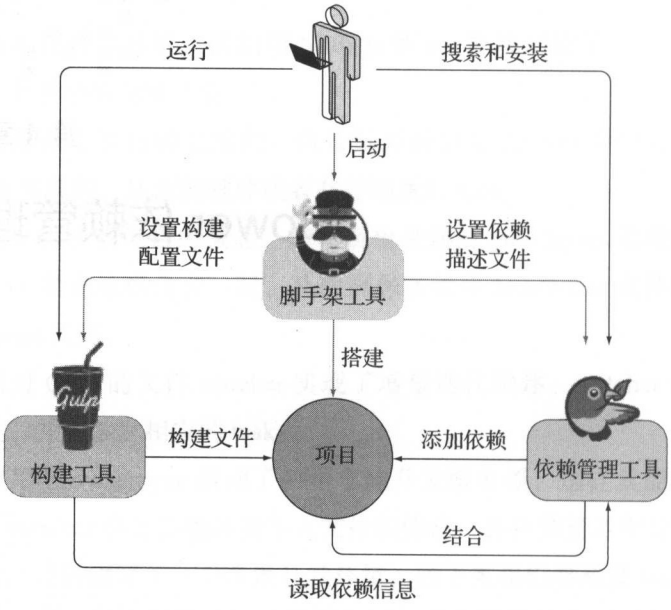



图 4-1 开发工作流的第二个工具：Bower，它是一个前端依赖管理工具。之后你将学会如何搜索和安装新的依赖，以及如何把它们添加到项目文件中去

在本章中，你将了解依赖管理工具是如何工作的，并且学会如何在自己的项目中使用它。在众多依赖管理工具中，我们选择了 Bower。它是特别针对前端开发者设计的。它有一个独特的设计，那就是在任何时候，项目中的每个依赖都只会有一个版本，从而确保浏览器中不会存在同一个库的多个版本。另外，比起别的依赖管理工具，它在处理版本问题的时候更加灵活，这一点对于 Web 要向后兼容的特点非常吻合。

首先，让我们来看看当谈论组件和依赖的时候，我们究竟在谈论什么。

### 使用 NPM 管理依赖

最近，NPM 和相关的 Yarn 也成为了前端开发者挑选依赖管理工具时的一项选择。但是，如果要在客户端用 NPM 管理依赖，就需要一个完全不同的工作流，而且这个工作流并不如 Bower 那么灵活。不过本章在写作过程中，也考虑到了其他的依赖管理工具。所以，读完本章后，读者应该能能轻松地将本章知识应用到其他依赖管理工具上。

## 4.1　依赖管理基础

在我们深入 Bower 之前，首先我们要了解依赖管理的基础。我们首先会定义什么是依赖，并且介绍依赖树的概念。再后面会介绍如何使用依赖管理工具来进行依赖管理。

### 4.1.1　组件和依赖

在整本书里面我们用来两个概念来描述可复用的代码包：*组件*和*依赖*。尽管这两者的内容是一样的，但是在管理它们的方式上还是有微妙的不同。

#### 1. 组件

组件是一段可复用的、独立完整的包，它自己能够实现一些功能。组件会提供 API 来调用它的功能。在本章中，我们会使用如图 4-2 所示的标志来表示组件。




图 4-2　我们使用这个图片来表示组件

组件的范围向上包括通用的类库，比如 jQuery（提供了跨浏览器的 API，让代码写起来更舒服），向下包括一些专门代码，比如 UI 组件。最重要的规则是要独立完整（self-containment）原则：一个组件的开发者必须保证，只要安装了这个组件，这个组件就能够正常工作，而不需要其他要求。这一点延伸出来下面的概念。

#### 2. 依赖

因为组件这个术语所代表的软件太多了，我们很快就会遇到一个问题：如果一个组件提供的功能需要其他组件才能运行怎么办？比如说一个轮播图组件，它其实是一个很专门的组件，它使用了 jQuery 来选择 HTML 元素。如果没有 jQuery，那么这个组件就无法满足最重要的独立完整原则。当然啦，你可以把整个 jQuery 库包括到组件中。但是 jQuery 本身就可以被视为一个组件，而且会被很多其他组件复用，所以应该把 jQuery 当成一个依赖。图 4-3 是本书用来表示依赖的标志。

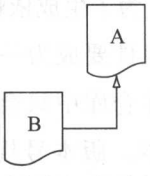



图 4-3　这个图片表示两个组件互相有联系。图里的箭头表示依赖关系。这个图片的意思是“组件 B 依赖组件 A”或者“组件 A 是组件 B 的依赖”

依赖是一个组件为了满足独立完整原则所必需的其他组件。依赖提供了这个组件所需要的功能，如果没有这些功能，这个组件就无法工作，

也就会违背独立完整原则。在图 4-3 中，组件 A 能够独立地为其他组件提供自己的功能，而组件 B 必须要和组件 A 一起才能工作。这层依赖关系对组件 B 是必需的，而依赖关系工具的任务就是确保满足这层依赖关系。

#### 3. 组件仓库

另一个需要定义的概念是组件仓库（component repository），或者叫 component registries。组件仓库是一个服务器，上面保存了所有登记过的组件的必要元信息。通常，依赖管理工具会组件仓库通信，来获取某个组件的元信息，包括版本、依赖和组件的下载地址等等。大部分情况下，这个组件本身也保存在组件仓库里。

然而对于 Bower 而言，Bower 组件仓库只会保存元信息。组件的文件本身则存储在其他 Git 仓库中，通常是 GitHub，这就是 Bower 的另一个优势。如果组件保存在其他地方，而不是保存组件元信息的仓库里的话，用户就能轻易地替换组件的下载地址。比如说，你可以不从 GitHub 下载，而从某个内网的 Git 仓库中下载。你还可以自己创建一个私有的组件仓库，来专门保存你自己的组件。

现在你已经知道了依赖管理工具是如何工作的了，接下来让我们看看依赖管理工具是如何管理组件之间的关系的。

### 4.1.2 依赖树

图 4-3 简单展示了两个组件之间的依赖关系，这种关系比较简单。但是实际上几乎每个组件都有多个依赖，而这些依赖自己可能还有别的依赖。这种依赖链可能会嵌套得非常深。不仅如此，不同的组件可能需要同一个依赖，但是要求的依赖版本可能不同。为了理清这错综复杂的关系，依赖管理工具会把所有的依赖构成一颗依赖树（dependency tree）。依赖树有三种：嵌套（nested）依赖树、扁平（flat）依赖树和两者的混合版。为了生成依赖树，所有的组件必须满足下面的要求：

一个组件要成为一个依赖，必须要有一个独一无二的名字，以和其他组件区分。比如，组件仓库中只会有一个名叫 jQuery 的组件。组件还必须提供版本号，用来区分不同版本。版本号规则体现的是 API 的兼容性（参见附录 A）。再以 jQuery 为例，Bower 的组件仓库中包含了好几个版本的 jQuery，从 1.0.1 版本到最新版本（比如说，3.0.0）。每个版本提供的 API 都有所不同，有可能会不兼容你的代码。只有知道了版本号，才知道用的是哪套 API。

满足这些要求后，接下来让我们深入探讨不同种类的依赖树。

### 1. 嵌套依赖树

嵌套依赖树是最直截了当的一种。图 4-4 展示了它的结构。

嵌套依赖树的基本理念是：每个组件各自都有自己的依赖，而不是共用一个依赖。这能够确保组件依赖的版本能够让组件正常工作。即使像图 4-4 里那样，有组件依赖了两个或多个版本的 A 组件，也不会产生冲突。

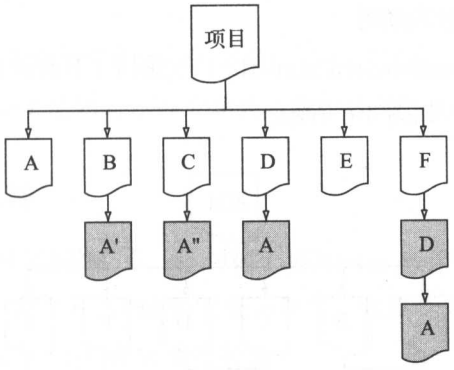



图 4-4　嵌套依赖树。组件 B、C、D 依赖组件 A，而它们各自都有一个 A 的副本。这保证了各个组件依赖的版本都能兼容这个组件（比如，B、C 需要另一个版本的 A，但是 D 依赖的还是原来这个 A）。组件 E 是一个独立的组件。组件 F 依赖组件 D，所以也间接依赖组件 A，D 和 A 都又被下载了一次。重复下载的组件在图里用阴影表示

另一方面，这会导致产生很多组件的副本，这些副本会让项目变得臃肿。如果一个组件两个或多个副本的版本不一样，那么有可能它们的 API 也有很大不同，组件的行为也会有很大不同。这就很混乱了。

### 2. 扁平依赖树

扁平依赖树和嵌套依赖树恰恰相反。图 4-5 展示了它的结构，这和图 4-4 是同一个项目。

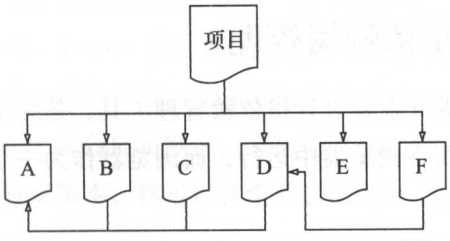



图 4-5　扁平依赖树。扁平依赖树保证了每个组件只有一个版本，如果有的组件依赖另一个组件（比如 B、C、D 依赖 A），它们就得使用同一个版本的组件

扁平依赖树中不会再有同一个组件的多个副本，每个组件在项目中只有一个副本。这保证了项目里不会有冗余的代码。但是扁平依赖树的缺点在于很容易产生冲突。如果两个组件需要同一个依赖的不同版本，那就会导致错误的发生，必须要用户自己来解决冲突。通常需要由用户来决定应该使用哪个插件，并解决潜在的不一致性。因为扁平依赖树正是 Bower 所采用的组件管理方式，所以下一节我们会深入探讨它。

### 3. 半嵌套 / 半扁平的依赖树

半嵌套或半扁平（semi-nested/semi-flat）依赖树（下面简称为混合依赖树）是上述两种依赖树的折中方案，如图 4-6 所示。

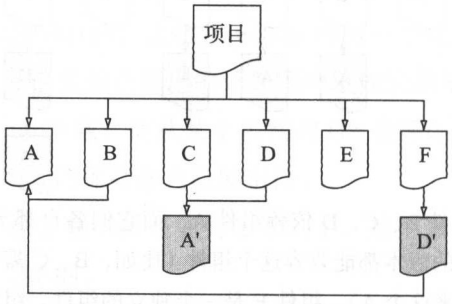



图 4-6 混合依赖树。如果有一个组件的依赖已经安装了，而且版本也兼容，那就不必重新再安装一次，只要指向已安装的那个包就可以了。如果版本不兼容的话，才会新安装一个版本兼容的包

混合依赖树的目标是用最高效的办法来管理一个包的不同版本，就像扁平依赖树一样。但是当出现冲突时，混合依赖树的做法就和扁平依赖树不同了，它会保留同一个组件的另一个版本。尽管这个办法看起来最合理，但是 Bower 之所以还是选择纯粹的扁平依赖树也是有原因的，下一节你就会知道。

## 4.2 Bower 的扁平依赖树管理

Bower 作为专为前端开发者设计得依赖管理工具，是完全基于扁平依赖树的。这是因为所有组件最后都要在浏览器中运行，而浏览器作为一个运行环境，和其他平台有着显著的不同：

- 浏览器有一个巨大的全局作用域，所有的组件都在里面运行。尽管有很多种办法可以创建作用域，但是大多数网站和应用还是把变量保存在全局命名空间里。

如果有两个不同版本的同名组件，那就会产生冲突。

- 依赖中的同一个组件的不同版本越多，需要传输的数据也就越多。由于开发者无法控制网络连接的速度和稳定性，所以每一个传输的字节都非常关键。前端开发者不能把代码变得臃肿。

由于这两个因素，Bower 采用了扁平依赖树。这也就是为什么我们采用 Bower 作为项目的依赖管理工具。

但是如你所知，这种方法很容易产生冲突，需要开发者尤其注意组件的版本和依赖。让我们来看一下 Bower 是如何在各个安装步骤中处理这些问题的。

### 4.2.1　安装和更新组件

由于之前我们已经用过 NPM 了，所以对于用 Bower 来初始化、安装和更新组件，应该不会陌生。Bower 的命令和 NPM 是完全一样了。如果你还没有安装 Bower，那就在命令行中输入下面的命令，用 NPM 来安装它：

```
$ npm install -g bower
```

安装好 Bower 后，就可以在命令行中使用 Bower 了。在执行下面的示例代码之前，先进入到一个空的目录中。为了让 Bower 初始化项目，你必须提供一个包含项目元信息的文件。这样一来，你的项目目录本身就成为了一个组件，并且保存了所有必要的依赖信息。

```
$ bower init
```

根据提示完成初始化后，就可以开始安装第一个组件了。图 4-7 展示了当你执行 bower install --save jquery 命令后会发生的事情。

save 参数的作用是把记录保存在元信息文件 bower.json 中。install 命令会下载 jQuery 的文件到一个叫作 bower_components 的目录中，这个目录就是保存所有组件和依赖的地方。当你执行这个命令时，Bower 会从 Bower 仓库中获取到组件的元信息，然后和 bower.json 中的组件的元信息进行比较。如果组件没有安装，那就会默认安装最新版本。如果已经安装了，Bower 就会自动停止，通知用户组件已经安装了。

如果你想安装某一个特定版本，你可以在组件后面添加上版本。比如：

```
$ bower install --save jquery#1.8.0
```

这个命令会安装 1.8.0 版本的 jQuery 到你的项目中。如果你想把 jQuery 升级到最

新版本，也很简单，输入：

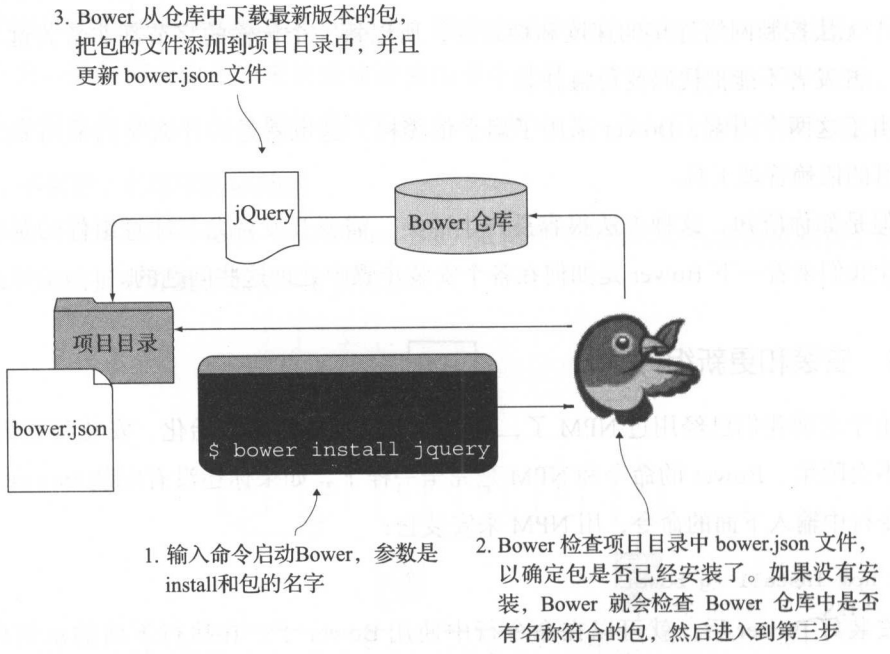



图 4-7 安装一个组件的过程。执行安装 jQuery 的级命令后，Bower 会下载最新版本的 jQuery，然后添加到项目中，并且在 bower.json 文件中会添加一条记录，包括了组件的名称和版本。更新组件的时候也会按照这个流程，区别在于更新时会用新文件替换掉旧文件

```
$ bower install --save jquery
```

就能重新走一次安装流程，强制 Bower 安装最新版本。

### 4.2.2 安装有依赖的组件

上述步骤都非常直截了当：安装一个组件，把这个组件变成项目的依赖。save 参数会把必要的元信息保存在 bower.json 文件中，把 jQuery 记录为一个依赖的组件。现在，让我们看看当我们安装一个自己有依赖的组件时，会发什么什么。我们这次以 Ember.js 为例，因为它正好依赖 jQuery，很适合当例子。这个例子不需要你对 Ember.js 有任何知识。首先，安装一个特定版本的 jQuery：

```
$ bower install --save jquery#1.8.0
```

然后安装 Ember.js，对上面的命令稍加修改就行了：

```
$ bower install --save ember
```

如果你的项目里还没有安装任何组件，Bower 会下载最新的 Ember 和最新的且兼容 Ember 的 jQuery。Bower 会把两个组件安装到项目目录中，同时在 bower.json 里保存必要的版本信息。不过在我们的例子中，已经有了一个 1.8 的 jQuery，于是就产生了一个特殊情况，如图 4-8 所示。

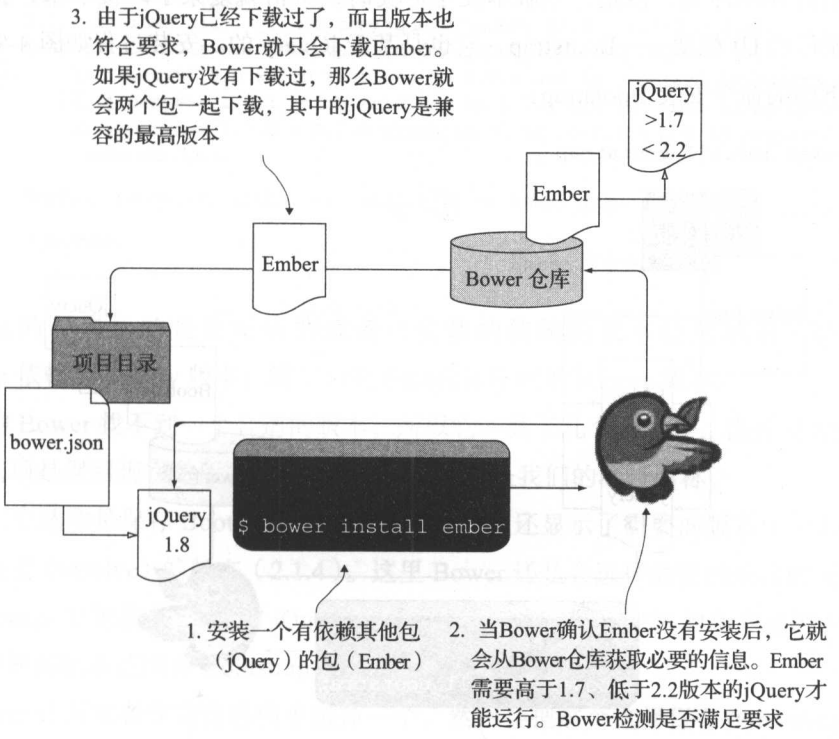



图 4-8　安装有依赖的组件的过程。Ember 的元数据说明 jQuery 的版本必须在 1.7 和 2.2 之间，这个信息会和项目中的 bower.json 里的信息比对。如果满足要求，那么就会开始下载这个包

Ember 在它的元信息文件里保存了它的的依赖的版本范围，其中包括 jQuery。根据元信息文件，Ember 依赖的 jQuery 的版本应该在 1.7 和 2.2 之间。Bower 拿到这个信息后，就会和 jQuery 的 bower.json 里的版本信息比对。现在我们的 jQuery 版本是 1.8，满足要求，所以会继续下载 Ember。

Ember.js 的作者特别指定了 Ember 所兼容的 jQuery 的版本范围，任何在这个范围中的 jQuery 都可以兼容。这一点是对于 Bower 是独有的，把 Bower 和其他类似的依赖

管理工具（比如 NPM）区分了开来（参见附录 A 以了解 NPM 是如何保证版本兼容性的）。但是这也需要组件的作者和维护者非常熟悉他们的组件和组件的依赖。

### 4.2.3 解决依赖冲突

到目前为止，我们还没遇到什么问题。我们安装的组件所需要的依赖，和项目里现有的组件并不冲突。但是，当版本发生冲突时，事情就复杂了。接下来，我们要安装的是流行的 UI 框架——Bootstrap，它也是基于 jQuery 的。安装过程如图 4-9 所示。

用下面的命令安装 Bootstrap：

```
$ bower install bootstrap
```

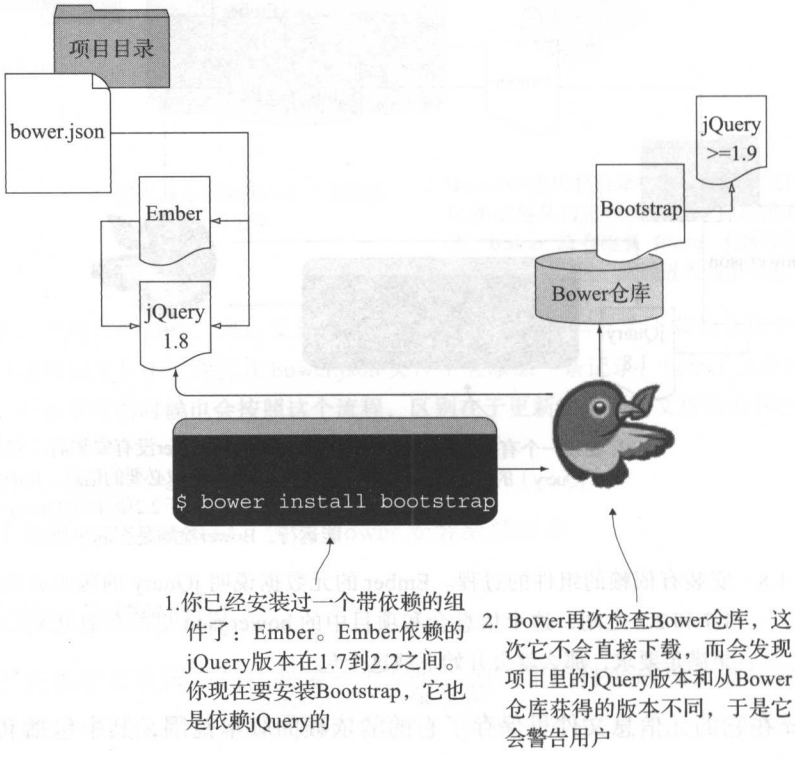
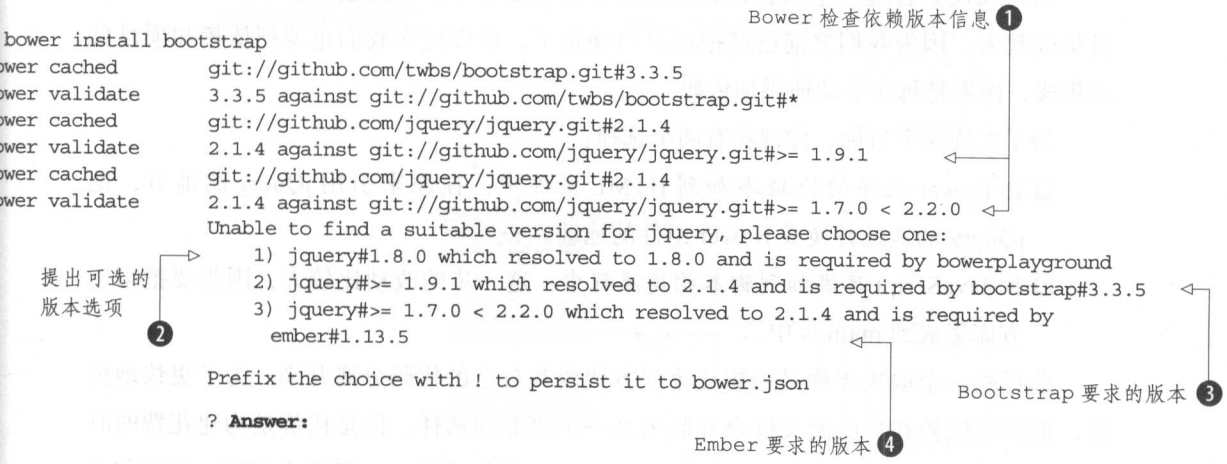



图 4-9 安装 Bootstrap 的过程和之前是类似的。区别在于，Bootstrap 需要的 jQuery 版本不同，且和已经安装的 jQuery 不兼容。Bower 会把项目中的 jQuery 版本和 Bootstrap 所需的 jQuery 版本进行比较，如果不同，Bower 就会抛出一个异常

在这个例子中，现有的 jQuery1.8 版本不兼容 Bootstrap 的最低要求 1.9 版本。Bower 自己不会处理这个问题，它会抛出一个异常，然后让用户来解决依赖冲突。下

面是命令行输出的内容：




❶这两行展示的是要安装的或者已安装的依赖的版本检测结果。第一行是 Bootstrap 依赖的 jQuery 版本；第二行是 Ember.js 依赖的 jQuery 版本。

因为 Bower 找不到一个合适的版本，所以它一共了几个可选项，供开发者抉择❷。第一个选项是保留现在的版本。bower playground 是我们的项目名称。

第二个选项是保存 Bootstrap 依赖的版本❸。它还显示了需要的版本（≥ 1.9.1）和最终会决定（resolve）的版本（2.1.4）。这里 Bower 还是会选择最新的兼容的版本。

Ember.js 也能兼容 2.1.4 的 jQuery，因为这个版本在元信息里的版本范围内。但是 Ember 依赖的版本范围和 Bootstrap 不一样。

Bower 让开发者在这些选项中选择一个，然后会把这个选择保存在 bower.json 里。如果你遇到了版本冲突的情况，一般最好先升级所有的依赖，看看你有没有过时的组件，然后再选择所有选项中决定（resolve）最多的那个版本，最后彻底地测试一次，看看有没有问题。

## 4.3　集成 Gulp

基于我们现在所掌握的概念，你应该已经能用 Bower 来管理之前的项目的依赖了。首先，从 scripts 目录中删除 vendor 目录。这个文件夹里（很有可能）有一个老版本的 jQuery，之后我们会用 Bower 来处理它。第二，在项目目录中执行 bower init 命令，把

它变成一个 Bower 组件。接着，用 `bower install --save jquery` 安装 jQuery。

如果仅仅只管理一个依赖，那么这些步骤就足够了。但是这些步骤没有和项目本身集成起来。因为我们之前已经把项目自动化了，所以这次我们也要把依赖和项目自动集成，而不是每次手动地添加依赖。

为了实现这个目标，你现在有两个选择：

- 把 Bower 组件的地址添加到 HTML 文件中：在原来引用 jQuery 的地方，把 jQuery 的地址替换成 Bower 组件的地址。
- 把 JavaScript 库添加到脚本构建流程中：这一步的改动比较大，因为要把第三方库集成到 main.js 中。

选择哪一个取决于你是否想让第三方代码和自己的代码分离开来。为了更快地传输，把所有依赖和自己的文件合并起来是一个更好的选择，但是代价是构建花费的时间更长，因为文件体积更大了。我们会在 Gulp 的进阶章节中处理这个问题。在目前看来，第一个选择能够让你迭代构建地速度更快。下面我们会讨论这两种选择。

### 4.3.1 用 Gulp 把依赖集成到应用中

对两个选择而言，你都要在 bower.json 添加一条记录。一般维护得比较好的组件都有这条记录，来帮助构建开发流：main-array。这个记录包括一个数组，这个数组里是一些文件名，这些文件是项目的主文件。因为很多组件都包含了很多你并不需要的文件（比如文档、主题之类的），只有一小部分文件是项目中要使用的。幸运的是，你可以通过很多插件来处理这件事。

首先，我们要把 Bower 组件集成到项目里，但用的不是之前那种把第三方代码文件地址写死在 HTML 文件中的写法，所以我们要修改一下 HTML 文件。一开始，先把下面这行代码从 index.html 中删除。

```
<script src="scripts/vendor/jquery.js"></script>
```

这行代码指向的是旧的 jQuery 模块，我们已经不需要了。为了把依赖注入到 HTML 文件中，我们要用一个特殊的插件，名叫 wiredep，这是“wireing up dependencies”的缩写。图 4-10 展示了 wiredep 的工作流程。

借助 wiredep，你可以在 HTML 文件中添加特定的记号，这些记号的作用是告诉插件在什么位置插入文件。wiredep 会把 bower.json 里列出的所有组件都加载一遍，然后找到相关的主文件，最后在正确的位置把这些文件的地址插入进去。

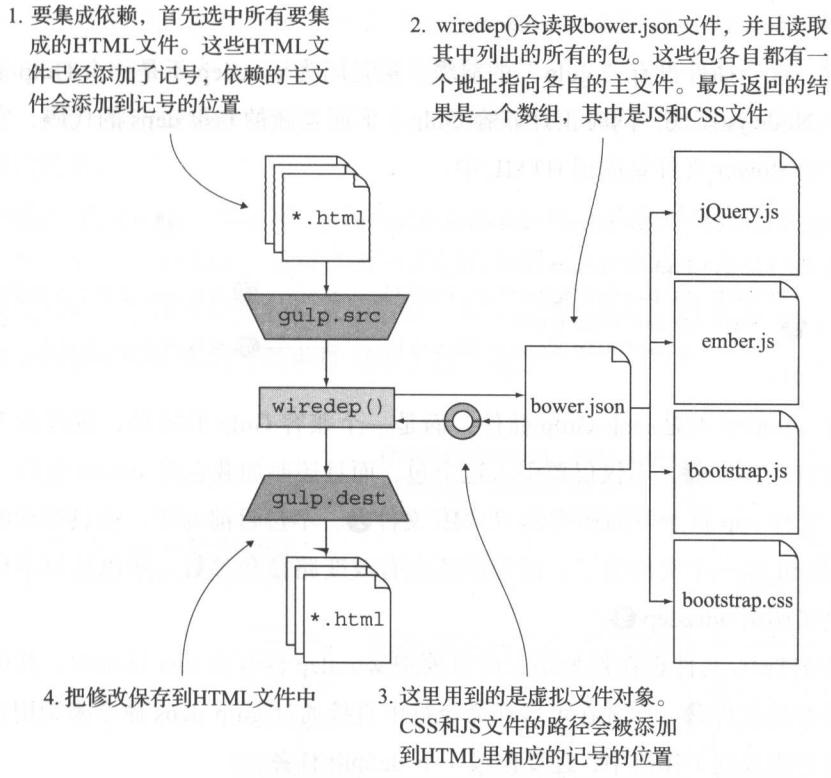



图 4-10　把 Bower 文件路径注入到 HTML 文件里的过程。wiredep 插件读取 bower.json 文件中的内容，获取到必要依赖的主文件的路径，然后把它们添加到开发者在 HTML 文件中标记的位置

首先，我们要修改 index.html 文件。在之前你引用 jQuery 的地方，以 HTML 注释的形式，加上如下记号：

```
    ...
    <!-- bower:js -->          <-- ❶ 注释开始
    <!-- endbower -->
    <script src="scripts/main.min.js"></script>   <-- ❷ 注释结束
</body>
</html>
```

wiredep 会在文件里面查找以 bower 开头的注释❶。冒号后面的是此处要引入的文件的类型。在这个例子中，就是 JavaScript 文件。endbower 的作用是告诉 wiredep 在什么地方停止❷。注意，所有在这两个注释之间的内容最后都会被 wiredep 删除。

这样一来，HTML 文件就已经准备好了，接下来要修改 Gulpfile 了。首先，把 wiredep 作为一个开发依赖安装：

```
$ npm install --save-dev wiredep
```

注意，这个插件没有“gulp”的前缀。这是因为 wiredep 不是一个 Gulp 插件，它只是一个 Node.js 模块，同时正好兼容 Gulp。下面是新的 task deps 的代码，它的作用是把主要的 Bower 文件集成到 HTML 中：

```
var wiredep  = require('wiredep').stream;          ❶ 引入 wiredep 包

gulp.task('deps', function() {
    return gulp.src('app/**/*.html')               ❷ 选择 app 目录下面所有的 HTML 文件
        .pipe(wiredep())                           ❸ 调用 wiredep
        .pipe(gulp.dest('dist'));                  ❹ 保存到 dist 目录中
});
```

由于 wiredep 不是一个 Gulp 插件，而是一个兼容 Gulp 的插件，所以在引入它的时候稍微有点不同❶。不仅仅要引入这个包，而且还要加载它的 stream 组件。在 deps task 中，选择 app 目录下面所有的 HTML 文件❷。不过目前为止，也只有你刚刚修改的 index.html 这一个文件罢了。因为你还没有改变路径和参数，所以可以不传入任何参数，直接调用 wiredep ❸。

由于 HTML 文件正在被处理，所以要把 wiredep 保存在 dist 目录中，其中包括了所有编译好的文件❹。你现在可以在命令行中直接通过 gulp deps 命令来调用这个任务了。要把它集成到工作流中，还要修改一下 default 任务：

```
gulp.task('default',
    gulp.series('clean',
        gulp.parallel('styles', 'scripts', 'deps'),   ← 把打包依赖的 task 和所有的资源相关的 task 并行执行
        'server',
        function watcher(done) {
            …
        }
    )
);
```

通过集成依赖文件，你已经成功地用新的管理第三方代码的方法替代了老的办法。现在只要用一个简单的命令（bower update jquery），就能平滑地升级 jQuery 依赖了。新的依赖也会自动集成到项目里，而不再需要特地手动操作了。

### 4.3.2 将依赖集成到 Gulp 构建流程中

还有一个办法可以集成新的 JavaScript 依赖，而不用把它们注入到 HTML 中。到目前为止，我们都在避免处理第三方的文件。对于测试和代码风格检查而言，这完全没有问题，因为没必要检查别人的代码风格。但是对于构建用于部署的代码而言，

我们应该考虑把第三方文件集成到构建流程中。这可以确保我们最后交付的是一个 JavaScript 文件，而不是多个，从而减少请求数。

通常，4.3.1 节中创建的 task 就够用了。但是在后面的章节中，还会在这个 task 的基础上进行优化。

在前面一节中，你已经了解了如何通过 wiredep 的 API 来轻松访问 Bower 文件。另一个插件 main-bower-files，也可以做到类似的事情。区别在于，它不会修改 HTML 文件，而是返回一个由文件路径构成的数组。你可以使用这个插件来为 scirpts task 创建更加复杂的 globbing 数组。图 4-11 展示了这一概念。

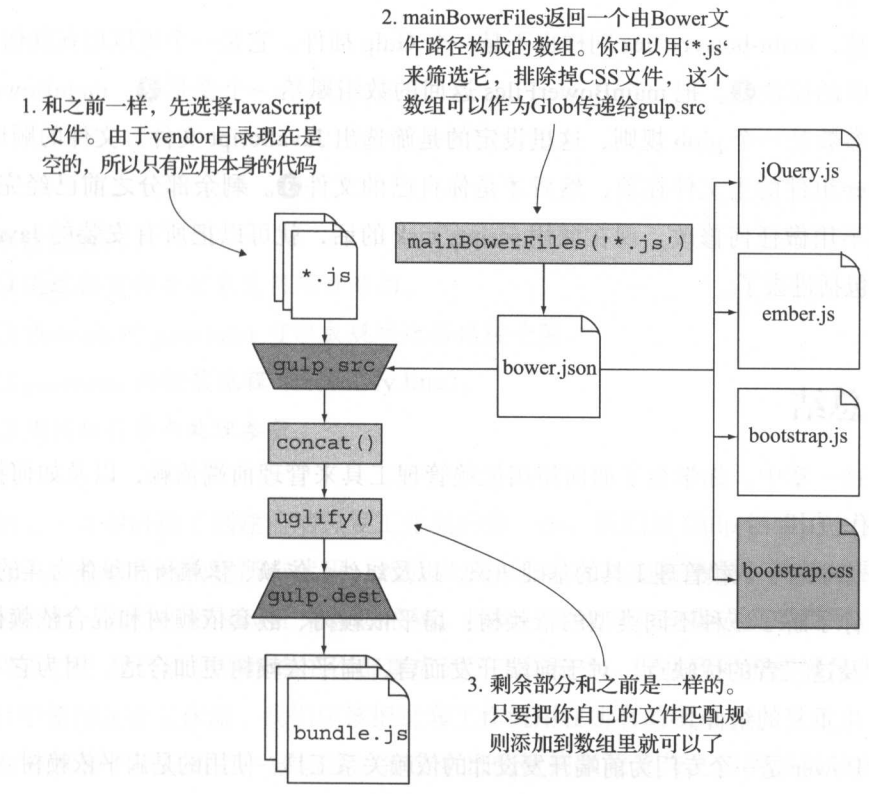



图 4-11　main-bower-files 内部做的事情和 wiredep 是一样的，但是它只会返回一个由文件路径构成的数组，你可以根据类型筛选这个数组。根据这些路径可以把相应依赖文件添加到构建管道中

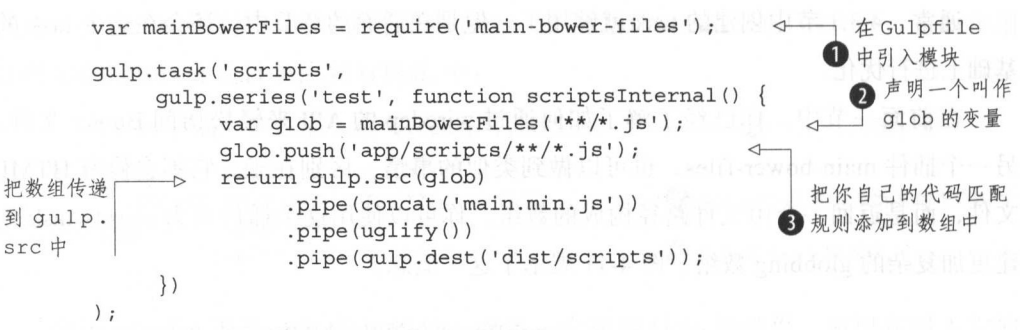
要使用这个插件，首先，你还是要在 HTML 文件里删除对 jQuery 的引用，剩下的部分在 Gulpfile 里完成。要安装 main-bower-files，就在项目的根目录里执行下面的命令

```
$ npm install --save-dev main-bower-files
```

安装完之后，就可以着手修改 Gulpfile 了。

```
var mainBowerFiles = require('main-bower-files');

gulp.task('scripts',
    gulp.series('test', function scriptsInternal() {
        var glob = mainBowerFiles('**/*.js');
        glob.push('app/scripts/**/*.js');
        return gulp.src(glob)
            .pipe(concat('main.min.js'))
            .pipe(uglify())
            .pipe(gulp.dest('dist/scripts'));
    })
);
```

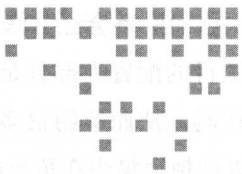



注意，main-bower-files 同样也不是一个 Gulp 插件。它是一个可以用在任何 Node.js 项目中的模块❶。把 mainBowerFiles 返回的数组赋给一个变量❷。mainBowerFiles 接受的参数是一个 glob 规则，这里设定的是筛选出 JavaScript 文件。文件的顺序是所有 Bower 组件的主文件在前，然后才是你自己的文件❸。剩余部分之前已经完成了，所以你不用做任何修改。现在调用 scripts task 的话，就可以把所有安装的 JavaScript 组件也包括进去了。

## 4.4 总结

在这一章中，你学会了如何使用依赖管理工具来管理前端依赖，以及如何把它集成到工作流中：

- 你学习了依赖管理工具的基础知识，以及组件、依赖、依赖树和组件仓库的定义。
- 你了解了三种不同类型的依赖树：扁平依赖树、嵌套依赖树和混合依赖树，以及这三者的优缺点。对于前端开发而言，扁平依赖树更加合适，因为它不会产生重复的组件。
- Bower 是一个专门为前端开发设计的依赖关系工具，使用的是扁平依赖树。Bower 通过和组件仓库通信，来决定是直接下载一个依赖，还是由开发者来做决定。
- 最后，你从 demo 项目中删除了所有第三方代码，并且创建还创建了一个流程，不仅用对应的 Bower 组件来替换掉原来的组件，还用 Gulp 把这个流程自动化了。
- 下一章中，我们会根据现在的 Bower 和 Gulp 的配置，让这个项目变得可以在很多新项目中复用。



第 5 章 Chapter 3

# 脚手架工具 Yeoman

**本章包括：**

- 通过搭建脚手架来复用项目结构。
- Yeoman 的 generator 可以成规模地搭建脚手架。
- generator 的组装流程（assembly line）。
- 用模板引擎来处理参数。

第 2 ~ 4 章讲述了创建前端开发工作流的第一步。我们用 Gulp 创建了一个本地开发环境，它能够自动构建所有代码，然后启动一个本地 Web 服务器，持续地展示代码更新的结果。我们还用 Bower 来管理依赖，把第三方库自动地集成到项目中。看起来（而且实际上）我们已经把很多工作都集成到了工作流中。但是我们不能满足于只在一个项目中使用这套工作流。我们应该把这套工作流用在所有的项目上。

在开发多个项目时，一般而言，每个项目的架构和配置与之前的项目基本上是类似的。这个现象很普遍，因为大家都不想重复造轮子。随着经验积累，你会知道在开发项目时要在哪里保存文件、新建哪些目录和依赖哪些组件。慢慢地，你会总结一种固定的模板，只要略加修改就可以套用到其他项目中。但是每次都重复创建这些文件和目录是很浪费时间的，我们希望把精力集中在每个项目的不同之处。这个创建过程完全可以自动化！这一步很关键。想象一下，在团队里，你的项目很可能不仅仅由你

一个人来做。你的同事会经常修改你的项目，你也会经常修改同事的项目。这个时候，如果你们的项目的配置大部分是一样的，只有很小的不同的话，那么你就能很轻松地接手别人的代码，从而节约很多时间。如果整个团队都使用同一套项目配置，那么团队成员就可以把精力集中在重要的任务上，而不会被基本的琐事拖后腿。

为了解决这个问题，你可以使用 Yeoman。第 1 章里已经提到，Yeoman 不仅可以创建项目所必需的目录、初始文件（比如构建配置文件）、模板代码，还能够安装依赖。Yeoman 能执行你之前创建的工作流，确保你和你的团队使用的是同一套项目配置。Yeoman 使用专门的、自动的 generator 来执行创建的工作，确保代码和项目结构不会产生任何意料之外的偏差。在本章中，你将学会如何创建一个 Yeoman 的 generator。这也是创建前端开发工作流的最后一步（见图 5-1）。

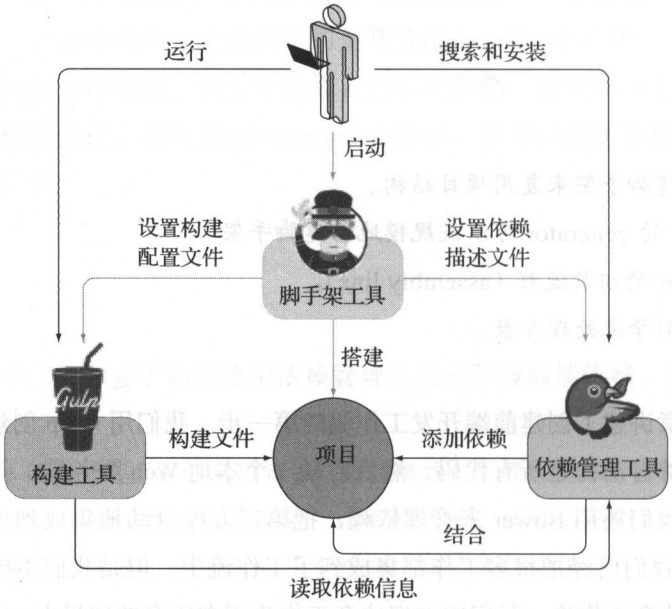



图 5-1 完整的开发工作流。通过 Yeoman，你可以复用其他项目的文件来创建一个新项目。这些文件包含了一个项目的基础架构，还有第 2、3 章的 Gulp 配置和第 4 章的 Bower 配置

## 5.1 Yeoman generator

当你开始一个新项目时，通常你会对项目架构有个初步构想。随着时间流逝，软件项目会不断膨胀，代码里会加入很多新文件和模块。但是如果你把一个已经开发了

一段时间的项目和一个初始项目相比较的话，你会发现它们依然遵循着一套共同的规则，如图 5-2 所示。

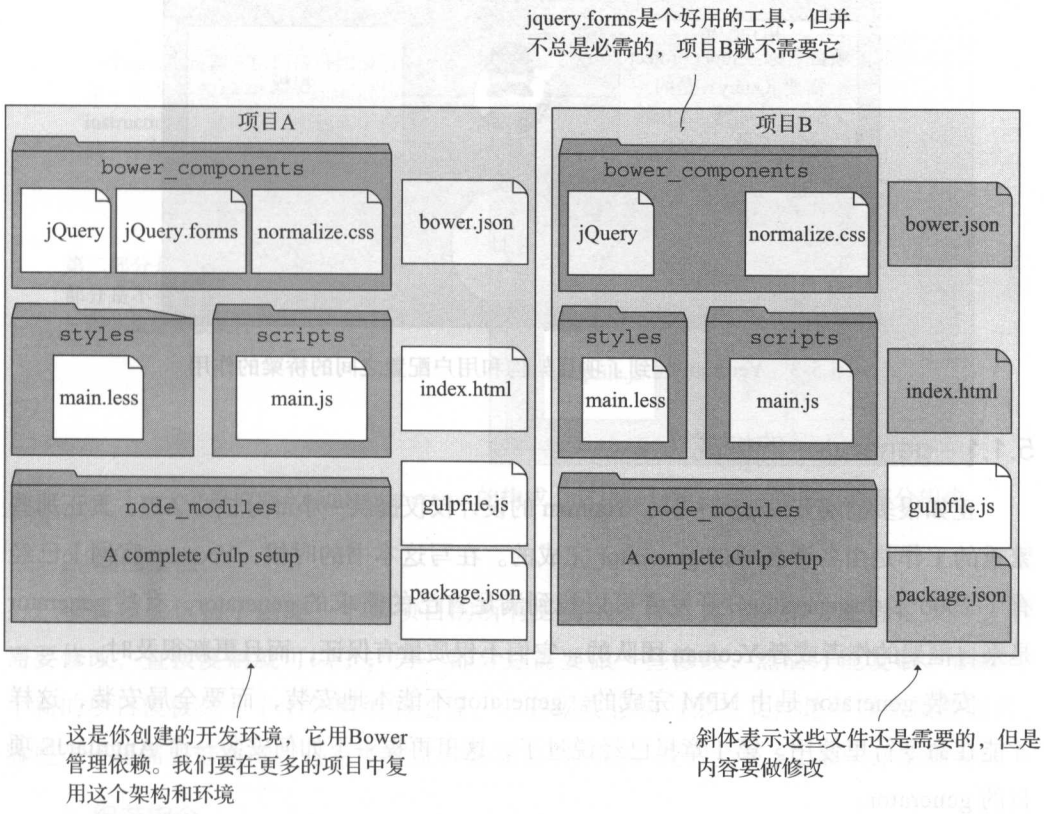



图 5-2　项目 A 和它的复制。项目 B 基本上就是项目 A 的复制，因此开发者可以立刻就开始写代码。但是项目 B 中的一部分有些不同。项目名字需要添加到 package.json、bower.json 和 index.html 中。jquery.forms 组件只有在需要的时候才要安装，不应该默认安装

项目 A 和项目 B 的目录结构是相同的，有些部分甚至完全一模一样，而另一些部分则会有细微差别，主要是文件内容的差异。这些细微差别很难维护，尤其是在把一个项目拷贝到另一个项目的时候，很有可能一不小心就被忽视了。这就是要把这个步骤自动化的原因。通过 Yeoman，可以使用 generator 来实现自动化。

Yeoman generator 是一个用来初始化项目配置和创建必要文件的工具。比如说，你想创建一个新的 AngularJS 的项目。你可以使用相应的 Yeoman generator，然后你需要回答一些问题，会有一些选项供你选择，比如要不要用某个库之类的。接着 Yeoman 会基于你的选择，修改好预设的模板，然后把结果交给你。最后，你会得到一个量身

打造的新项目。图 5-3 展示了这个流程。

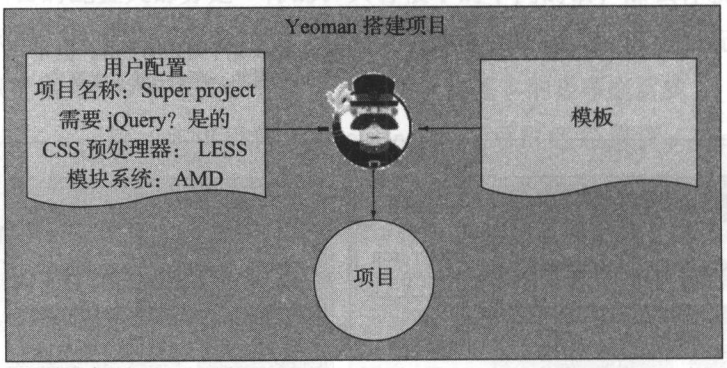



图 5-3 Yeoman 起到了项目架构和用户配置之间的桥梁的作用

### 5.1.1 generator 的构成

正如很多前端开发工具一样，Yeoman 的设计仅仅提供一小部分核心 API，真正那些繁重的工作是由各种各样的 generator 完成的。在写这本书的时候，Yeoman 官网上已经有了 2000 多个 generator。开发者可以找到满足自己的需求的 generator，有些 generator 是来自框架的作者或者 Yeoman 团队的。它们不仅质量有保证，而且更新很及时。

安装 generator 是由 NPM 完成的。generator 不能本地安装，而要全局安装，这样才能在命令行里使用。第 1 章里已经说过了，这里再提一下如何安装一个 AngularJS 项目的 generator。

```
$ npm install -g generator-angular
```

AngularJS generator 中包括了项目文件、依赖和一个推荐的项目架构。因此，generator 不仅给 Yeoman 提供了新项目的模板文件，还提供了组装它们的指令。图 5-4 展示了总体的架构。下面，我们来看一些注意事项。

#### 1. 名称

按照约定，Yeoman generator 的名字必须以“generator-”的前缀开头。原因是因为 generator 都是普通全局安装的 Node.js 模块（见附录 A），所以 Yeoman 完全依赖文件系统来找到它们。Yeoman 会在所有的模块中查找所有的以“generator-”开头的模块，然后再根据 generator 的 package.json 文件中的元信息找到要找的 generator。5.4 节我们会深入研究。

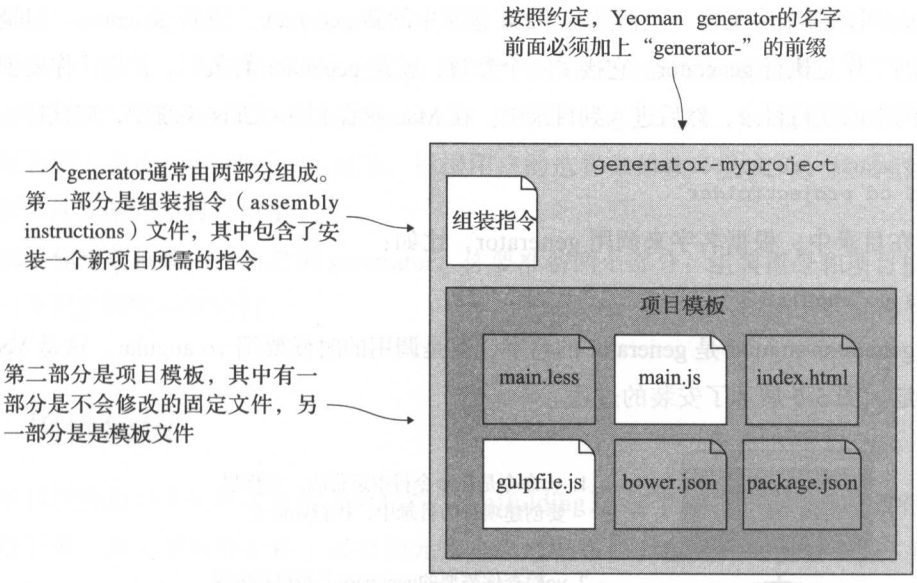



图 5-4　一个典型的 Yeoman generator 的构成，它由项目模板和组装指令两部分构成

### 2. 项目模板

项目模板包括了创建一个新项目的所有必需的文件。其中一部分是固定文件，不需要修改，直接复制就可以了；另一部分是需要做一些修改，然后才能复制。创建一个新的项目模板是一个十分精细的过程。你必须要想明白哪些文件是可以直接复制的，哪些文件是需要修改的，还有哪些文件是需要引用的。我们会在 5.2 节中详细研究。

### 3. 组装指令

Yeoman generator 里有一个文件的作用是告诉 Yeoman 如何搭建项目，它负责给用户提供安装提示和选项，让用户来选择。它还知道每个文件应该放在哪里，做哪些修改。组装指令会在 5.3 节中介绍。

## 5.1.2　调用 generator

执行 generator 需要在命令行中用 yo 命令执行。yo 是 Yeoman 的一个命令，起到的是 generator 和项目目录之间的桥梁作用。你可以把它理解为 generator 的执行器，yo 知道怎么找到 generator，也知道怎么执行它。由于它是基于 Node.js 的，而且必须要在要搭建脚手架的项目目录里调用，所以 yo 必须要全局安装。在安装的时候要加上 -g 参数：

```
$ npm install -g yo
```

yo 可以做很多工作，包括浏览 NPM 仓库中的新 generator、更新 generator，但是它最主要的工作是执行 generator。它接受一个参数，就是 generator 的名字。首先，你要创建一个新的空的项目目录，然后进入到目录中。在 Mac 或者 Linux/Unix 系统中，应该这么做：

```
$ mkdir projectfolder
$ cd projectfolder
```

在目录中，根据名字来调用 generator，比如：

```
$ yo angular
```

generator-angular 是 generator 的名字，但是调用的时候要用 yo angular。这是 Yeoman 的约定。图 5-5 展示了安装的过程。

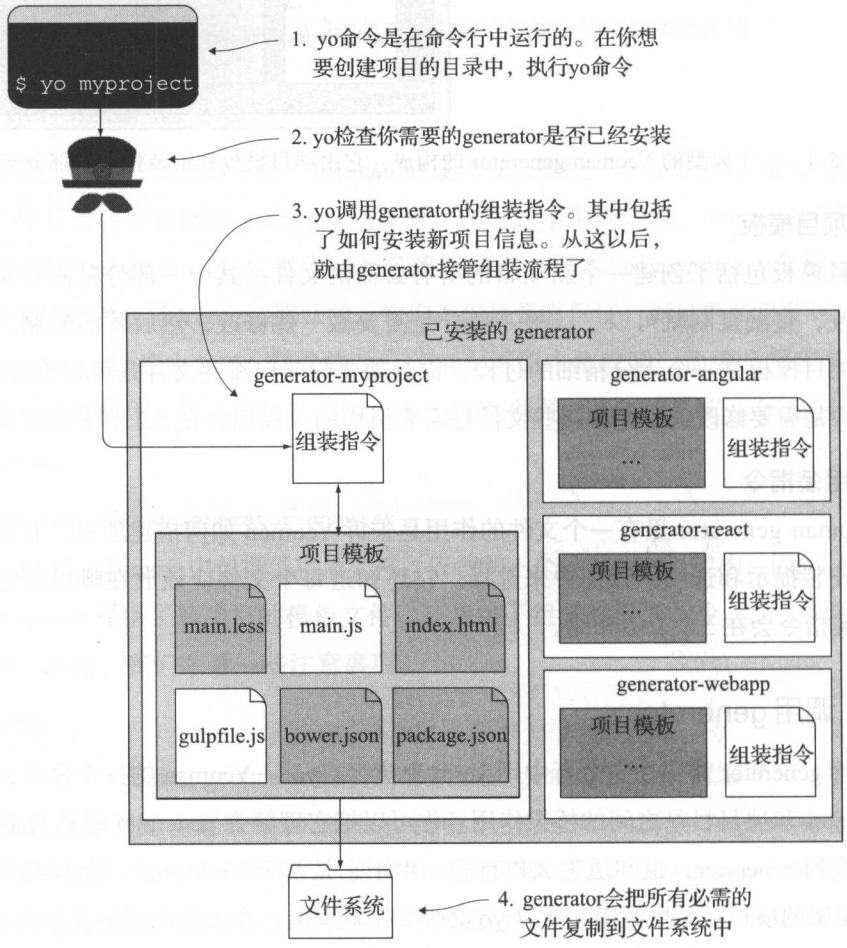



图 5-5 Yeoman 搭建脚手架的过程

yo 命令行工具首先会检查 generator 是否可用，如果是，它就会调用组装指令。generator 会接管之后的组装流程，把文件加以修改并且复制到文件系统中。

yo 的主要功能是找到你需要的 generator 并且调用它。之后，yo 就会执行组装指令。组装指令会向用户询问一些信息，根据用户的选择来修改一些文件，然后把所有的模板文件复制到项目目录中。

如果你想要创建一个自己的 generator，你要准备两个部分：组装指令和项目模板。让我们分别来研究一下它们。

## 5.2　项目模板

项目模板也经常被称之为脚手架材料（scaffolding），这个术语来来源于建筑行业，比喻很形象：脚手架材料提供了必要的元素来创造新东西。但是你可能会疑惑，脚手架材料是由什么东西组成的呢？

为了决定哪些脚手架材料是必需的，你要研究你的项目，找出其中共同的元素，从而形成一套项目模板。项目模板中的每个文件都应该有不同的作用。有些文件是必需的，而且不能做修改；有些文件是空白的，以后可以往里面添加内容；还有些文件应该足够灵活，这样在每次创建新项目时都能够对其中的一部分进行修改。

最好是把这些文件分门别类，这样当你要创建一个新项目时，就知道该如何处理这些文件了。图 5-6 里展示的是图 5-2 里的两个项目经过抽象后的核心部分。

图里展示了三类文件：固定文件（加粗）、灵活文件（flexible parts）（斜体）和可选文件（普通字体）。所有在阴影区域中的元素都不是项目模板的一部分，它们都是可恢复文件。恢复这些文件的所有信息都保存在其他文件里（package.json、bower.json）。你可以用命令来恢复它们，这样就可以把项目模板变得尽可能小。

观察一下这个项目里的每个元素，你会发现它们可以被分为四类：固定文件、灵活文件、可选部分和可恢复部分。前三个部分将会被添加到项目模板中：最后一类文件（正如名字所示）会在搭建脚手架的时候恢复。

### 5.2.1　固定文件

固定文件是在每个项目里都一样的文件。它们不仅内容一样，而且是项目所必需的。缺少这些文件，项目的脚手架就不能算是搭建好了：

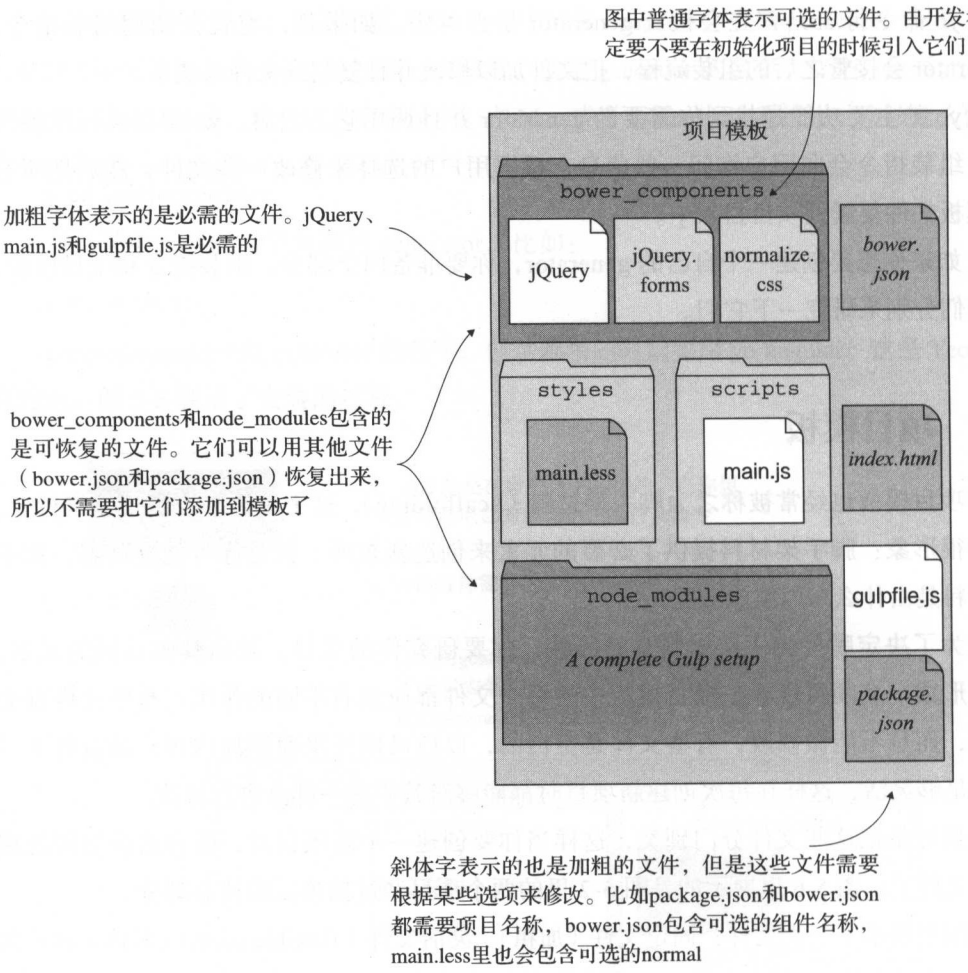



图 5-6 一个项目模板

- 用 Gulp 和所需的 Gulp 插件来构建系统：在项目架构允许的条件下，本地开发环境设计得越灵活越好，因为这样你就可以复用这些文件，而不用修改项目了。
- 核心的库和依赖：在示例项目中，jQuery 是创建应用所必需的。在其他项目中，可能 AngularJS、Ember.js 等其他组件是必需的。
- 有的文件初始状态是一样的，但是之后会向其中添加内容：相比于灵活文件，固定文件在搭建脚手架的时候不需要任何修改。以 main.js 文件为例，它在一开始的时候是空的，只有几行代码，封装了一些模块。你会在之后往里面添加内容来实现你的应用，但是每个项目一开始的时候，这个文件都是空的。

你的项目模板在引入这些文件的时候不需要任何修改。

### 5.2.2　灵活文件

灵活文件和固定文件有相似之处，因为它们都是搭建一个项目所必需的，但是灵活文件的内容会根据不同项目而改变。这些改变可以很细微，也可能很巨大。让我们看两个例子：

- 每个项目都有不同的项目名称。bower.json 和 package.json 都需要项目的名称才能工作，而且在 index.html 里也要把项目名称添加到 title 属性里。名称就是一个变量，每次搭建脚手架的时候都需要改变。所以 bower.json、package.json 和 index.html 都是灵活文件，这就是所谓的细微的改变。
- 有些可选文件（见下一节）会对其他的文件造成直接影响。比如说，引入 normalize.css 文件（它是一个保守 CSS 的 reset，作用是使所有浏览器的默认样式保持一致）就会直接影响到 main.less，因为必须要在 main.less 里引用 normalize.css 才行。根据文件的数量不同，这类改变可能会相对比较巨大。

在所有项目模板文件中，灵活文件是首先会受到用户影响而改变的文件。因为在搭建脚手架的过程中，需要用户输入参数来决定项目最终的样子。

### 5.2.3　可选文件

正如之前提到的，可选文件并不是搭建脚手架时所必需的元素，但是有时候它能锦上添花，给项目提供一个良好的默认设置：

- 之前提到的 normalize.css 就是一个很好的项目默认设置，因为它能够让浏览器以同样的样式展现 HTML 元素。大多数情况下，你都会把这个文件引入到项目中。但是如果你想用一个 CSS 框架，比如 Bootstrap，那么 normalize.css 就显得多余了，因为 Bootstrap 已经有了。这就是 normalize.css 是可选文件的原因。
- 其他组件也是如此，比如图 5-3 中的 jquery.form。如果你有一堆表单要处理，那么用它很合适，但是有时候它只是白白增加了项目的体积。用不着的代码就应该从项目中删掉。

根据处理项目的方式不同，可选文件可能会很多。另一方面，如果你只需要很少的组件，或者你对自己必需的组件非常熟悉，那么可选文件就可以变得很少。

### 5.2.4 可恢复文件

可恢复文件是项目模板中最重要的元素之一：你不需要引入所有文件。你可以把项目里的某些部分当成黑盒来使用，只调用它们的 API，而不在乎它们内部的实现。Gulp 用来创建本地开发环境的包就是很好的例子：你只管安装 gulp-less，并不在乎包里面的内容。你只要正确引入了这个包，整个应用的功能就不会受到影响。我们完全可以让 Bower 来为我们做这件事。具体的步骤就是：下载必需的组件，然后把它们安装到项目中，最后引入它们。依赖管理工具有一个非常强大的功能：只要文件的元信息完整就把原来的文件恢复出来。由于 package.json 和 bower.json 保存了所有依赖的信息，所以你不需要自己引入这些依赖，只要引入这两个文件就可以了。在 package.json 文件的目录下，执行 npm install 就能下载 package.json 里的所有包。在 bower.json 的目录下执行 bower install 也是一样的。所以完全没有必要引入依赖。

这些分类能够帮助你使用 Yeoman generator 创建一个项目模板。好消息是你不需要把可恢复文件引入到项目模板中，因为之后可以直接下载它们。固定文件可以不经思考地就直接复制。灵活文件和可选文件就需要额外的关注了。Yeoman generator 里负责处理这些文件的是组装指令。下一节我们就会来研究组装指令，在 5.4 节中会创建一个项目模板和安装文件（assembly file）。

## 5.3 Yeoman 的组装流程

准备好项目模板后，就要开始准备组装指令了。组装指令是用来让 Yeoman 创建项目所需文件的一系列具体的命令。要执行这些命令（bring some order to those instructions），Yeoman 要按照预先定义好的一套组装流程（assembly line）一步一步工作。组装流程由三个连续的步骤构成，这三个步骤由组装指令构成。图 5-7 是三个步骤的概览。

这三个步骤分别是：

- 安装提示（prompting）：通过向用户提问，来让用户自定义配置。
- 写入文件（writing）：从项目模板中把文件复制到目录中，这是搭建脚手架最关键的一步。
- 安装依赖（installing）：必需的文件都复制完毕后，就要安装本地开发环境所必需的 Bower 依赖和 node 模块。

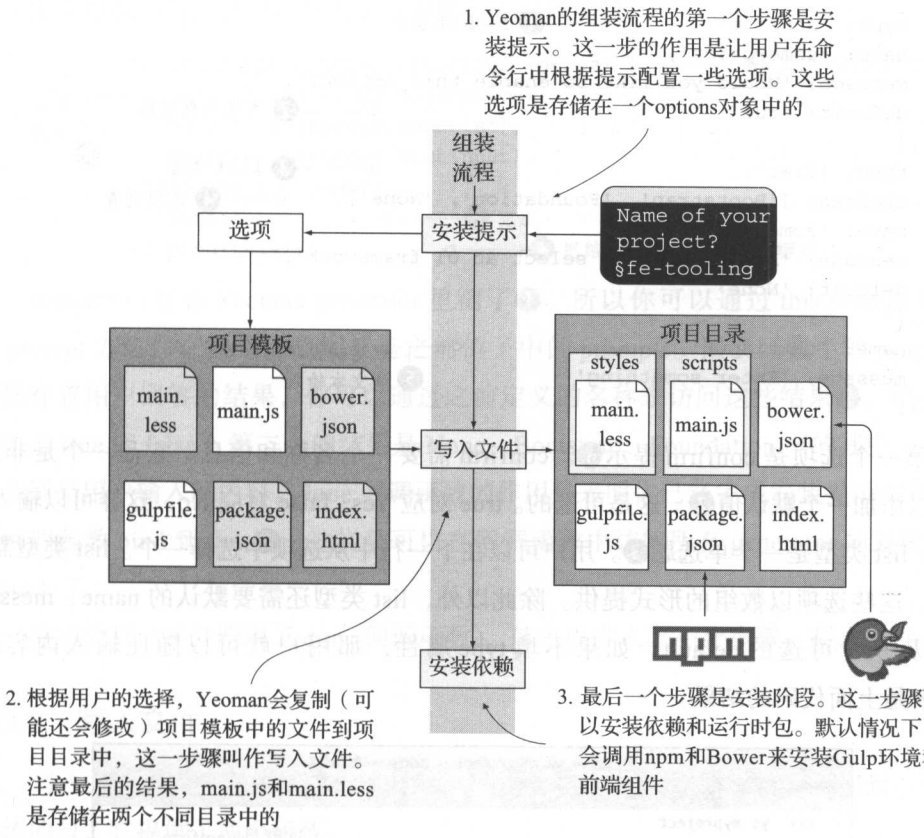



图 5-7　典型的 Yeoman 组装流程分为三个主要步骤。第一个步骤是安装提示，用户通过选择提供的选项，或者直接输入配置，来完成模板。第二个步骤是写入文件，根据用户的选择，项目文件会被复制到指定的目录中。第三个步骤是安装依赖。这一步会下载并安装所有保存在 bower.json 和 package.json 中的第三方库。最后本地开发环境就大功告成了

### 5.3.1　安装提示

安装提示是 Yeoman 组装流程的第一步，也是需要用户和 generator 交互的最后一步，后面的流程就完全是自动化的了。安装提示通过直接在命令行中询问用户，来获得 generator 所需的参数。

Yeoman 首先会执行安装提示步骤作为整个组装流程的第一步，你需要提供一些选项来组成你的安装提示。为此，你需要使用一个名叫 inquirer 的包。这个包的作用是生成选项让用户选择。请看下面的代码：

```
var prompts = [{
```

```
    type: 'confirm',                          ❶ comfirm 提示
    name: 'someOption',
    message: 'Would you like to enable this option?',
    default: true                                    ❷ 可选的默认值
  },{
    type: 'list',                                    ❸ list 类型
    choices: ['Bootstrap', 'Foundation', 'None'],    ❹ 选项列表
    name: 'someList',
    message: 'Do you want to select an UI framework',
    default: 'None'
  },{
    name: 'textinput',                        默认的提示是简单
    message: 'Enter something'                ❺ 的文本输入
  }];
```

第一个选项是 confirm 提示❶。confirm 需要一个名称和信息，它是一个是非题。你可以添加一个默认值❷，这是可选的。true 对应 Yes；false 对应 No。这样可以输入得更快。list 类型是一个单选题❸。用户可以在下一行中从选项中选择一个。list 类型需要选项，这些选项以数组的形式提供。除此以外，list 类型还需要默认的 name、message 属性及一个可选的 default。如果不填 type 属性，那用户就可以随便输入内容❺。图 5-8 是上面代码的结果。

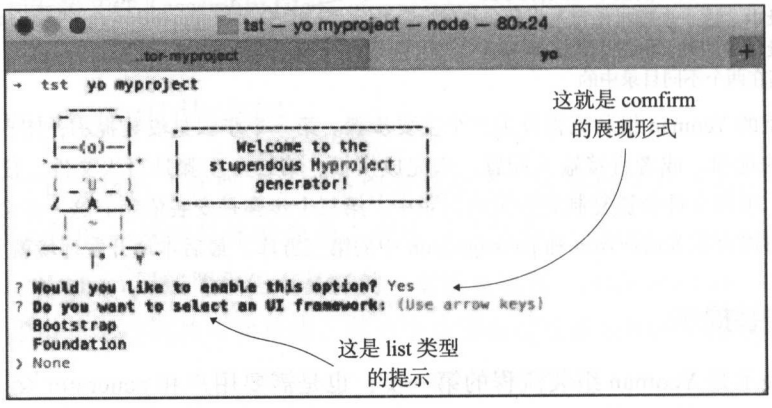



图 5-8 Yeoman 中的 comfirm 和 list 提示的输出结果的示例

name 和 message 是每个提示都需要的字段。name 是到时候保存结果的对象的属性名称，所以一定要记得添加 name 属性，message 是提示信息。



注意 想要了解全部的提示类型，请查看 inquirer 的文档：https://www.npmjs.com/package/inquirer。

定义好提示后，就可以提问了。需要一个回调函数：

```
this.prompt(prompts, function (props) {   ← ❶ 调用 prompt 方法
  console.log(props.someOption);          ← 访问结果 ❷
  console.log(props.someList);
  console.log(props.textInput);
this.props = props;
  done();                                 ← ❸ 完成
}.bind(this));                            ← ❹ 把结果和 generator 绑定
```

inquirer 已经在 Yeoman generator 里面了❶。所以你可以通过 this.prompt 直接调用 prompt 方法，它接受的参数就是之前例子中的 prompts。回调函数中的 props 参数中保存着用户回答的结果，你可以通过之前定义的名称来访问这些结果❷。第一个是结果是 true 或 false；第二个的结果是 None、Bootstrap、Foundation 中的其中一个；第三个则是用户输入的内容。done 回调函数的作用是表明你已经完成安装提示步骤了❸。把回调函数 bind 到 this ❹。这样就可以在回调函数中访问所有 generator 的属性（this.props）了。

完成之后（并且调用了 done 回调函数），就可以进入下一个步骤：写入文件。

### 5.3.2　写入文件

写入文件步骤会把项目模板复制到指定的目录中。为此，Yeoman 要用到一个文件系统的包，它提供了两个函数：

- fs.copy：把一个文件复制到另一个指定的目录中。
- fs.copyTpl：作用和上面的相同，但是会事先经过模板引擎。

fs.copy 更简单一些。图 5-9 展示了它们是如何工作的。

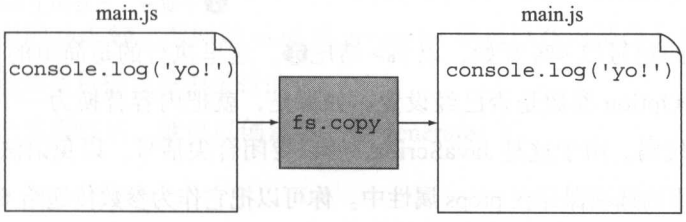



图 5-9　通过 fs.copy 把一个文件复制到另一个地方的方法，这个方法会在指定的目录创建一个一模一样的文件

这是 Yeoman 复制文件的方法，和你在命令行或者文件管理器中的操作差不多。

典型的调用应该像下面这样：

```
this.fs.copy(
    this.templatePath('_package.json'),
    this.destinationPath('package.json')
);
```

调用你想要复制的 fs 对象

可以访问项目模板中的文件

定义好输出文件的路径，通常是调用 generator 的目录

第二个方法 fs.copyTpl 更加高级。图 5-10 展示了它是如何工作的。

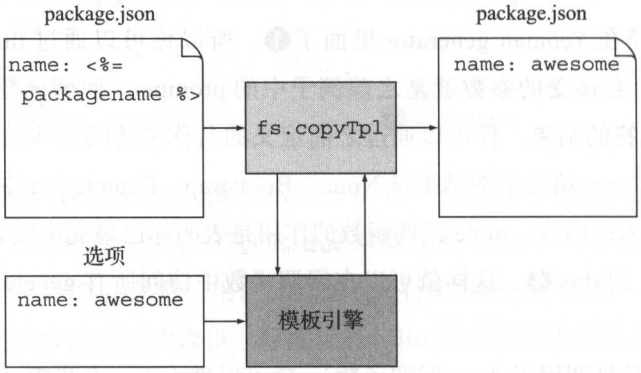



图 5-10 fs.copyTpl 更加高级一点。它不只是复制文件，它还会首先把文件用模板引擎解析一遍。模板引擎会解析文件中的特别记号，然后把占位符替换为真正的内容。模板引擎从 Yeoman 直接获得选项

这里你可以使用上一节定义好的安装提示。模板引擎会解析文件中的特别标记，然后替换成别的内容。比如说，如果你只想在某个参数（someOption）已经设置好的前提下引入某些部分，你就可以这么来写 bower.json：

```
"dependencies": {
  "jquery": "~2.1.4"     <% if(someOption) { %>,
  "bootstrap": "~3.3.5" <% } %>
}
```

❶ 设置 someOption 参数了吗？

❷ 如果设置了，结果就是在上一行出现一个逗号，然后加上 bootstrap 的这行

模板引擎的记号以 <% 开始，以 %> 结尾❶。这里执行的是简单的 JavaScript，作用是检查 someOption 参数是否已经设置。如果是，就把内容替换为一个逗号，再加上 bootstrap 这行代码。由于这是 JavaScript，所以要闭合尖括号，以免语法错误。

你想要调用的选项保存在 props 属性中。你可以把它作为参数传递给 fs.copyTpl 函数：

```
this.fs.copyTpl(
   this.templatePath('bower.json'),
   this.destinationPath('bower.json'),
   this.props
);
```

调用 Yeoman 的 fs 对象；前三行是一样的。

保存属性的对象


注意　Yeoman 所用的模板语言是 EJS。更多信息参见 http://www.embeddedjs.com/。

写入文件步骤会通过 fs.copyTpl 来处理可选文件和灵活文件，并且通过 fs.copy 来处理固定文件。接下来的步骤就是处理可恢复文件。

### 5.3.3　安装依赖

Yeoman 组装流程的最后一个步骤就是安装阶段。现在目标目录里已经什么都有了，只需要调用 npm 和 bower 来下载并安装依赖和 node 模块就行了。对于这个步骤，Yeoman 也提供了一个方法来调用：

```
this.installDependencies();
```

就这么简单。这个方法会调用 Bower 和 NPM（如果可能的话）并且安装所有 package.json 和 bower.json 里提到的组件。如果你只想安装其中一个的话，Yeoman 还分别提供了针对 Bower 和 NPM 的方法：

```
this.npmInstall();
this.bowerInstall();
```

你还可以通过 this.spawnCommand 调用你自定义的安装命令。比如说，如果你想等到所有 Bower 依赖一安装好，就用 Gulp 把它们打包起来，你就可以用下面这个方法。

```
this.installDependencies(function() {    <-- ❶ installDependencies 接受一个回调
  this.spawnCommand('gulp', ['deps']);    <-- ❷ 通过 spawnCommand 调用 gulp
}.bind(this));                            <-- ❸ 把回调 bind 到 generator
```

通过这个方法你可以在 NPM 和 Bower 安装完依赖后执行其他函数❶。deps 参数会调用第 4 章中的 task ❷。把回调 bind 到 generator 就可以在内部执行所有的函数❸。所有这些命令都是在目标目录中进行的。

完成这三个步骤以后，就可以创建第一个 generator 了。

## 5.4　创建一个 generator

Yeoman 社区提供了很多 generator。但是如果你做的项目用的不是什么特定的技术，那么你很可能对于 generator 要怎么搭建有着自己的想法，所以你需要创建一个自

己的 Yeoman generator。有了前面几节的知识，你可以轻松地在三个步骤内完成它。这一次，你要离开项目目录，在其他文件夹里工作。随便给文件夹起个名字，然后你就可以开始创建项目模板了。

### 5.4.1 准备项目模板

对于项目模板，你需要从示例项目中挑选一些文件，然后把它们变得更灵活，这样才能继续。你可以随便在哪个目录下做这件事，反正之后会把它们移动到一个特定的位置。图 5-11 展示了项目模板的组成。

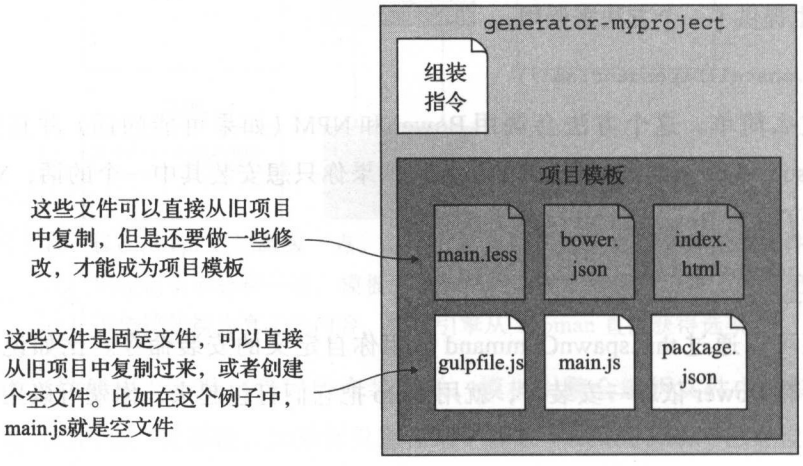



图 5-11 generator 架构的示意图，包括了组装指令和项目模板

项目模板中的所有文件都是必须要从旧项目中复制过来的文件。你的模板应该由下面这些文件构成：

- 固定文件：main.js——一个空的 JavaScript 文件，你要创建一个新的 main.js 文件。
- 固定文件：package.json——从第 3 章的 Gulp 设置中复制而来。
- 固定文件：gulpfile.js——这个文件也已经有了。
- 灵活文件：main.less——如果你安装了 normalize.css，那就把它引入到 main.less 中。否则，main.less 也是空文件。
- 灵活文件：bower.json——如果你安装了 normalize.css，那就把它添加到 bower.json 中。
- 灵活文件：index.html——这个文件包含项目名称。

创建一个新目录，然后把上述文件复制进去。对于main.js和main.less，就创建两个空文件。不需要创建子目录，因为之后会用generator创建。



注意 如果你觉得麻烦，你可以在Github下载示例模板：https://github.com/frontend-tooling/sample-template。

打开bower.json，然后添加下面的代码：

```
{
  "name": "sample-project-gulp",
  "version": "0.0.0",
  "license": "MIT",
  "ignore": [
    "**/.*",
    "node_modules",
    "bower_components",
    "test",
    "tests"
  ],
  "dependencies": {
    "jquery": "~2.1.4" <% if(includeNormalize) { %>,
    "normalize.css": "~3.0.3" <% } %>
  }
}
```

总是引入jQuery；如果设置了includeNormalize，就引入Normalize

这一行引入了normalize.css

在main.less中，添加下面的代码：

```
<% if (includeNormalize) { %>
@import (css)
     '../bower_components/normalize.css/normalize.css';
<% } %>
```

LESS的引入文件的命令

这个LESS的引入文件的命令可以引入任何CSS文件，这个路径是相对于样式文件目录的。

最后要修改index.html。打开index.html，添加下面的代码：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title><%= projectTitle %></title>
    <link rel="stylesheet" href="styles/main.css">
</head>
<body>
    <!-- bower:js -->
    <!-- endbower -->
    <script src="scripts/main.min.js"></script>
</body>
</html>
```

这一行的内容会被替换为projectTitle，如果设置了的话

注意这个模板用的标记是“<%=”而不是“<%...”，这个标记的意思是告诉模板引擎直接用这个属性替换内容。

修改好这三个文件后，灵活文件也准备完毕了。bower.json 和 packge.json 中保存了所有可恢复文件，在这个例子中，normalize.css 是可选的。

## 5.4.2 创建组装指令

凭空创建一个 generator 需要花费很长时间。幸运的是，Yeoman 团队坚持脚手架的理念，提供了一个用来创建其他 generator 的 generator。听起来有点绕，其实用起来很简单。要安装这个 generator，只需要在目录下运行下面的命令：

```
$ npm install -g generator-generator
$ yo generator
```

根据安装提示回答文件。Yeoman 会一步一步引导你把你的 generator 发布到社区，但是我们目前并不打算这么做，所以无视掉这一部分。当你完成后，配置就完成了。把模板复制到 generator/app/templates 目录下，然后开始编辑 index.js 文件。这个文件里包含组装之间。如你所见，其中已经包含了三个阶段的代码，只要做一些修改就行了。

### 1. 安装提示步骤

你可以很容易就找到 prompting 的调用。它已经包括了一个简单的安装引导，你可以对它进行简单的修改。下面代码中加粗的部分是要修改的地方：

```
prompting: function () {
  var done = this.async();

  // Have Yeoman greet the user.
  this.log(yosay(
    'Welcome to the stupendous ' + chalk.red('Myproject') + ' generator!'
  ));

  var prompts = [{
    type: 'confirm',
    name: 'includeNormalize',
    message: 'Would you like to include normalize.css?',
    default: true
  },{
    name: 'projectName',
    message: 'Enter the name of your project'
  }];

  this.prompt(prompts, function (props) {
    this.props = props;
    done();
  }.bind(this));
},
```

决定是否引入 Normalize 的参数；这是一个是非题

一个代表项目名称的字符串，之后要被添加到 index.html 中

把 props 复制到 generator 对象中

修改完了以后你在 generator 对象中就有了一个新对象——props，接下来让我们进入到写入文件步骤。

### 2. 写入文件和安装依赖步骤

写入文件步骤也已经在 generator 里预先设置好了，只要做一些修改就行了。和上面一样，只要修改加粗部分：

```
writing: {
  fixed: function () {                          ❶ 用函数填充 writing 对象
    this.fs.copy(
      this.templatePath('package.json'),
      this.destinationPath('package.json')
    );
    // Same for gulpfile.js
    this.fs.copy(
      this.templatePath('main.js'),
      this.destinationPath('scripts/main.js')   ❷ 可以指定子目录
    );
  },

  flexible: function () {                       在 flexible 函数里要执行所有 fs.copyTpl 调用
    this.fs.copyTpl(
      this.templatePath('index.html'),
      this.destinationPath('index.html'),
      this.props
    );
    // bower.json, main.less 也是如此
  }
},
```

这些函数❶里可以做任何事情。为了使架构更清晰，你要区分 flexible 和 fixed 函数[⊖]。fixed 函数会执行所有 copy 操作。在这个例子中❷，会在目标目录中创建一个 scirpts 目录。

完成写入文件步骤后，剩下的事情就是安装依赖了。代码已经到位了。如果你想要修改，你可以参照 5.3.3 节在其中调用 Gulp。

## 5.4.3　构建和测试

现在项目模板和组装指令都已经准备好了，你可以试着使用 generator 了。要检查

⊖ 读者可能会发现，现在的 generator 中的 index.js 已经采用了 ES6 的 class 语法。而在上面的代码中，作者使用的是 ES6 之前的写法。他把 generator 中的 writing 属性设置成了一个对象，在对象中写函数。这种写法在采用了 ES6 写法的 generator 中已经不适用了，如果在 ES6 语法中直接复制上面的代码，是会报错的。读者可以把上面代码中 writing 对象中的每个函数内容，写在一个函数中，并把这个函数当作 writing 方法。这样达到的效果和原本是一样的。——译者注

generator 能不能工作，在要搭建脚手架的目录里面执行下面的命令：

```
$ npm link
```

这个命令的作用是让你不用全局安装或者发布，就能全局调用 generator。如果你想让别人也能用你的 generator，你有两个办法：

- 发布到 NPM：首先你要在 npmjs.com 注册，然后在 generator 的目录下执行 npm publish 命令。这样一来就能在 yeoman.io 和 npmjs.com 中找到你的 generator 了。
- 推送到一个 Git 仓库：如果你想要一个私有的 generator（可能是因为你们公司的保密政策），你可以把文件放在一个 Git 仓库中，你的同事可以用 Git URL：`npm install -g git://path/to/your/repository.git` 来安装。

不管你的选择了那一方法，现在你和你的同事都可以用它来搭建项目的脚手架了。

## 5.5 总结

本章主要讲述了什么是脚手架，以及如何用 Yeoman 搭建脚手架：

- 首先我们了解了 Yeoman generator 的构成：项目模板和组装流程。我们还学会调用 Yeoman 的命令行工具 yo 来使用 generator。
- 我们知道了什么是项目模板和它的构成：固定文件、灵活文件、可选文件和可恢复文件。固定文件不需要改动，灵活文件需要修改，可选文件可以选择要不要，可恢复文件通过其他文件重新创建。了解这四类文件后，就知道一个真实的项目中的哪些文件需要被添加到项目模板了。
- 说到组装流程，Yeoman generator 通常包括三个步骤：安装提示、写入文件和安装依赖。每一个步骤都关系到一个 Yeoman API。我们还了解了 Yeoman 所使用的模板引擎。
- 最后，你把之前的示例项目做了一些修改，然后提取项目模板和组装指令，使用 Yeoman 的 generator-generator 创建了第一个 Yeoman generator。
- 这一章是第一部分的最后一章。我们的工作流到此就已经能够工作了。但是我们仅仅只用到了一些基本技术。在第二部分，我们会更加深入地研究我们所使用的技术，创建更加高级的构建配置文件、generator 和 Bower 组件。

第二部分 *Part 2*

# 工具链的集成和扩展

- 第 6 章　不同环境下的 Gulp
- 第 7 章　使用流
- 第 8 章　扩展 Gulp
- 第 9 章　创建模块和 Bower 组件
- 第 10 章　Yeoman generator 进阶

有了第一部分的基础，接下来是时候深入研究这三个工具的技术了。在第二部分，你将会用更多的工具来扩展工具链，并且从中得到更多的好处。

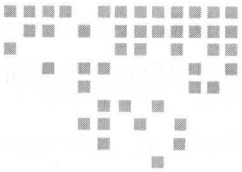
第 6 章中，你将会通过引入增量构建和 sourcemap 来改进本地开发环境。你还会学习如何根据不同的部署环境来编译出不同的代码。

第 7 章会深入挖掘 Gulp 的底层技术——文件流。通过使用流，不仅可以避免代码的重复，还可以创建可复用的构建管道。

第 8 章展示了如何通过 Gulp 插件以外的工具来扩展 Gulp，你将会学习 Promise 和 Gulp 的虚拟文件系统。

第 9 章回过来讨论 Bower 和组件的概念。其中展示了如何把组件分成模块，以及如何用模块来更高效地把组件集成到项目中，你还会学习如何发布自己创建的 Bower 组件。

第 10 章中，你会学习什么是 Yeoman sub-generator，以及如何通过它来自动创建新的模块。



第 6 章 Chapter 6

# 不同环境下的 Gulp

**本章包括：**

- ❑ 增量构建和构建缓存。
- ❑ 通过 sourcemap 调试源文件。
- ❑ 切换开发环境和 noop。

在本书的第一部分里，你已经学会了如何创建一个强大而健壮的构建配置，我们把它称之为本地开发环境。这套配置能够自动运行那些繁琐却又普遍的任务，连开发服务器也能够自动地更新每次迭代的构建。

如果你仔细观察构建的产物，你会发现它们都是可以直接部署的代码。JavaScript 和 CSS 都已经被优化和压缩，依赖也都已经打包，随时都可以部署到生产服务器上。这么做的一个巨大优点是可以确保真正部署的代码和在本地测试和构建的代码是一样的。当然，这么做也有缺点。想象一下你在应用里发现了一个 bug，JavaScript 不运行了，控制台里会报错。但是你的代码是压缩过的生产环节的代码，你看到的代码很可能就跟图 6-1 里的一样。

构建过的代码已经被混淆得面目全非了，即使是高手也无法找到发生错误的地方。所以有的人可能就会开始猜，试图找到出错的那一行，并且对应到源文件，然后猜测错误发生的地方。这就会导致第二个问题：每次修改都需要把代码完整地构建一次。

每个文件都要读取、代码检查然后压缩。根据应用的大小，构建可能需要好几秒钟，这会极大地拖慢定位错误的时间。

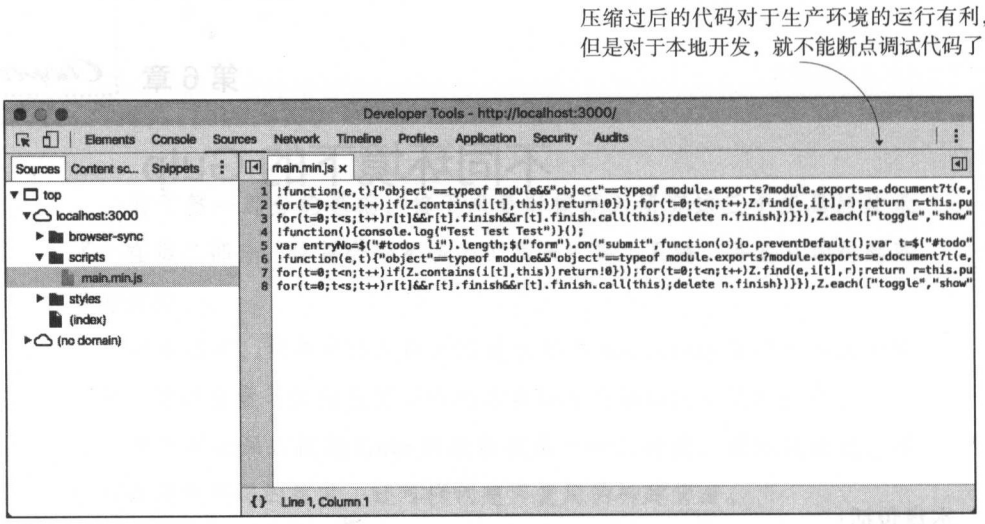



图 6-1 压缩和合并过后的 mian.js 文件在 Chrome 开发者工具中的样子。代码已经面目全非，完全看不出来原来的痕迹

如果能够在源文件中看到代码出错的那一行，就可以解决这个问题。如果还能够仅仅只重新构建修改的这一行，而不是重新构建整个应用，就能大大加快构建速度，更快地在浏览器里看到修改的结果。而且最好还能够在保留现有构建流程的前提下，实现这件事。最后还要再加上一些新工具，使其能够在开发环境和生产环境之间切换，避免遗漏构建生产环境的代码。

其实这些事情都能通过配置 Gulp 的选项来实现：

- 你可以使用增量构建（incrementalbuild）并将中间结果缓存起来，从而加快每次迭代构建的速度。
- 跨浏览器的标准 sourcemap 可以让浏览器在开发者工具中显示源代码，但是实际运行的还是编译和压缩过后的代码。这能够帮助你在生产环境下定位代码的错误。
- 开关（switch）可以根据要构建的环境（生产或开发环境）来决定是否采用 sourcemap 和增量构建。

在本章中，你将学会如何使用这些技术让开发环境变得更加高效，同时兼具优雅和性能。

## 6.1　增量构建和构建缓存

我们的 Gulpfile 的原始设计是每当文件系统里的一个文件改变时，就执行所有相关的 task。每次你保存一个 JavaScript 文件，就会执行 scripts task，接着再执行 test task。图 6-2 再现了第 3 章中的这个流程。

我们之前的流程是每次迭代后，都要选择所有 JavaScript 文件，读取它们的内容，然后进行处理。这个流程固然好，但是真的有必要在每次保存一次文件的时候都对所有文件进行 JSHint 检查吗？还是只检查改动的文件？显然后者更好，因为其他文件之前都已经检查过了，没必要再检查。这里就应该让每次迭代都增量构建，意思是说每次迭代都只构建新的或修改的文件，然后合并到之前的构建结果中。让我们来看看应该如何实现。

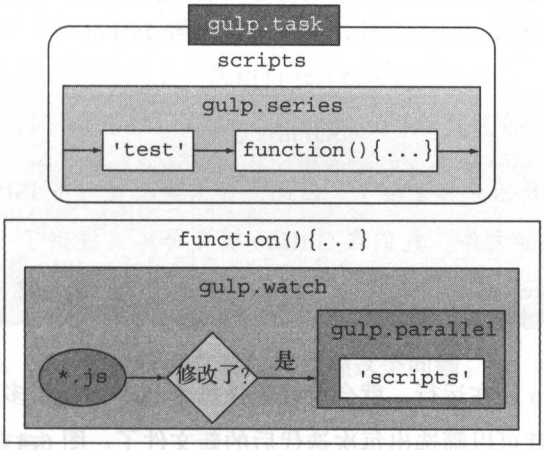



图 6-2　第 3 章内容再现。最上面的是 scripts task，它首先执行 test，然后开始编译 JavaScript。下面的是 watcher，监听 JavaScript 文件的变化

### 6.1.1　Gulp 内置功能

Gulp 刚诞生的时候就有了增量构建的功能。人们从 Gulp 控制和操作文件流的能力中，认识到了 Gulp 的强大之处。幸运的是，随着 Gulp 4 的发布，Gulp API 新增了一个非常方便的功能，即根据时间戳来选择文件。让我们来看看怎么把这个功能用在我们的 test task 上。图 6-3 展示了第 2 章介绍的 task。

每当你调用 test task 时，它都会选中所有以 js 结尾的文件，然后对它们运行 JSHint。现在我们想让这个选择文件的过程更加智能。Gulp 提供了一个叫作 since 的选项，可以作为一个参数的属性传递给 gulp.src 方法。它接受一个时间戳作为参数，并

且检查所有选择到的文件是否符合这个时间戳。如果所有的文件都比这个时间戳要旧，那所有的文件都不会被选中。

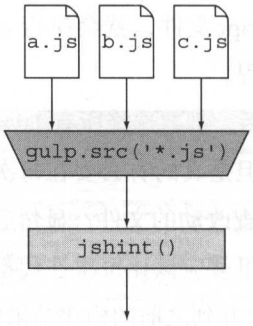



图 6-3 这是原始的 test task 的流程。每当它执行时，它都会选中所有 JavaScript 文件，并把它们传递给 JSHint

### JSHint 设置

你的应用比一开始更加复杂了。因此，你需要配置一下 JSHint，告诉它要检查哪些文件、忽略哪些文件。我们在 GitHub 项目仓库里提供了一个 .jshintrc 配置文件。在 sample-chapter-gulp 项目里切换到 ch7 分支，就能看到这些改动。

另外，只要 Gulp 还在运行，就会持续记录每个 task 运行了多少时间。有了这个信息和 since 选项，你就可以筛选出每次迭代后的新文件了。图 6-4 展示了这个过程。

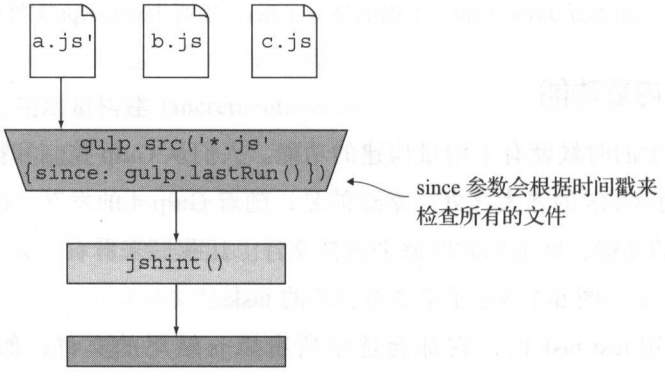



图 6-4 借助 since 参数，你可以根据时间来筛选匹配的文件。增量构建会匹配自从上次执行任务后修改过的文件，这就是为什么这个方法的名字叫作 lastRun

gulp.lastRun 接受一个参数：task 的名称。如果没有传递参数给它，Gulp 就会返回上一个运行的任意 task 的时间戳。注意 Gulp 只会持续记录它在每次运行时执行任务的时间戳。如果 Gulp 停止运行了，那么之前记录的时间戳就丢失了。下面是代码实现。

**代码清单 6-1　Gulpfile.js 节选：选择增量的文件**

```
gulp.task('test', function() {
  return gulp.src(['app/scripts/**/*.js',
    '!app/scripts/vendor/**/*.js'],
    { since: gulp.lastRun('test') })
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
    .pipe(jshint.reporter('fail'));
});
```

gulp.src 的第二个参数是一个 JavaScript 对象。

这个 JavaScript 对象给选择文件的过程添加了一些约束。在这个例子中，我们使用了 since 选项，它的值被设置成上一次运行 test task 的时间戳。

减少每次迭代的读取操作和 JSHint 的运行次数可以显著加快代码风格检查的速度。如果你想创建一个生产环境（也就是说不需要迭代）或者仅仅只是想运行 test task，也没有关系，你还是可以匹配到所有的文件。

上面的代码工作得很好，但仅限于对于那些每次从源目录输出到目标目录的文件数量相同的 task 而言。如果流程中包含更多文件转换的 task，比如 scripts task，上面的代码就有些局限了。接下来让我们看看如何处理这个问题。

### 6.1.2　安装构建缓存

尽管内置的增量功能对于单文件操作工作得很好，但是我们无法在 scripts task 中不做任何修改就使用它。增量构建对于 scripts 是必需的，因为 Uglify 过程是比较消耗性能的。如果仅仅只压缩那些新文件或者修改的文件，就能大大地减少运行时间。但是我们最后输出的结果是一个文件 main.min.js，这是应用里所有 JavaScript 文件编译后的文件。所以尽管 Uglify 操作可以只针对部分文件执行，但是在最后合并文件的时候还是会用到所有的文件。如果在选择文件时用了 since，那最后就无法选择到所有的文件。

这里就要用到构建缓存（build cache）技术来解决这个问题。构建缓存会在内存中保存所有曾经选中过的文件。记住，当你第一次运行任务的时候，每个文件都是新的。Gulp 社区通过 gulp-cached 插件提供了构建缓存的功能，它的算法很简单，如图 6-5 所示。

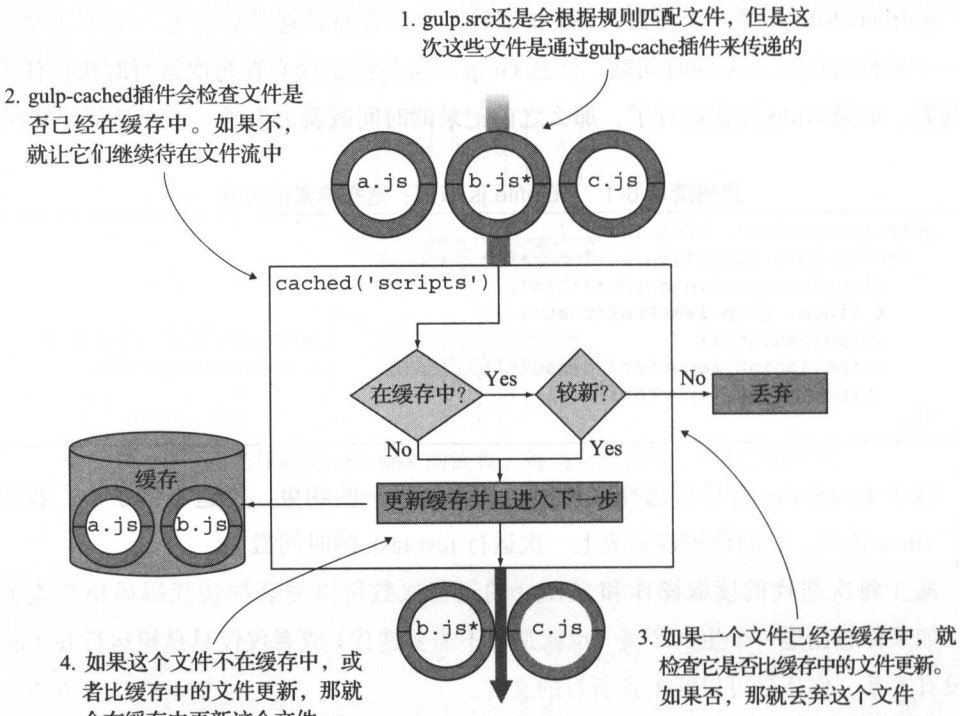



图 6-5 gulp-cached 插件的流程。首先，匹配所有的原始的 JavaScript 文件。插件会检查文件是否已经在缓存中。如果否，那这些文件就会被传入下一步，然后添加到缓存中。如果它们已经在缓存中了，插件就会检查它们是否比缓存中的文件更新。如果是的话，它们就会被传入下一步，并且更新缓存。如果不是，它们就会从文件流中被丢弃

这一过程看起来是一个更高级的 since。那些已经处理过并且没有修改过的文件会被过滤掉，这样可以减少文件流中的文件数。但是它和 since 有个重要的区别：since 不会识别文件的内容。如果一个文件根据时间戳判断是一个更新的文件，它就会被添加到文件流中。同样的，如果你使用了 gulp-cached 插件，你还是要通过 gulp.src 来读取文件，所以它并不能减少读取的操作。

所以最好的做法是把它们结合起来。构建缓存能带来的最大的好处是，你需要已经处理过的文件时，可以直接使用缓存里的文件。通过结合使用 gulp-remember 插件，你就从可以把之前缓存的构建好的文件重新添加到文件流中。图 6-6 展示了这个过程

借助 gulp-cached 和 gulp-remember，你可以实现下面这些优化：

- 你可以读取之前没有读取过的文件。
- 你可以检查文件内容的差异。

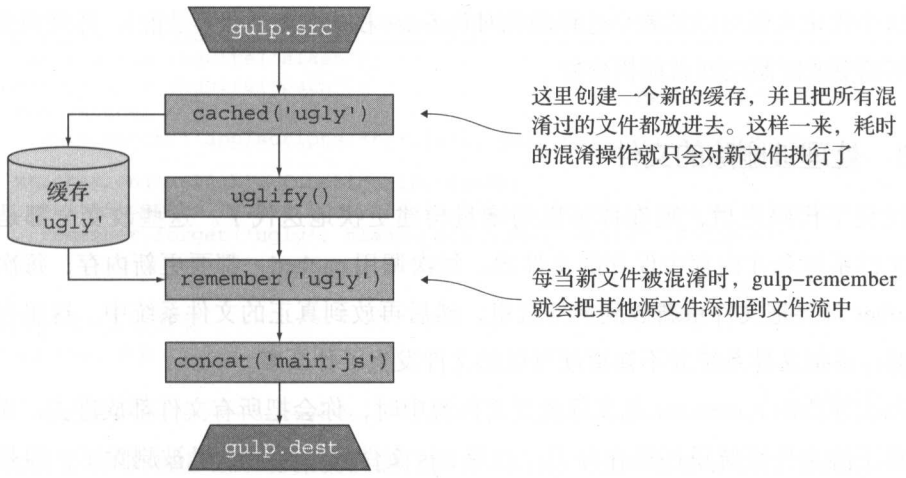



图 6-6　gulp-remember 让你可以“反过滤”文件流。所有原始的文件都会添加到被 gulp-cached 过滤过的文件流中。这样一来，就可以减少像 Uglify 这样耗时的操作所花的时间了

- 你可以处理之前没有处理过的文件。
- 你可以重新使用之前缓存过的文件。

下面的代码是 Gulpfile 的实现，但是首先你要用 NPM 安装 gulp-cached 和 gulp-remember 插件。

**代码清单 6-2　Gulpfile.js 节选：Gulp 缓存**

```
...
var cached      = require('gulp-cached');
var remember    = require('gulp-remember');
...

gulp.task('scripts',
  gulp.series('test', function scriptsInternal() {
    var glob = mainBowerFiles('**/*.js');
    glob.push('app/scripts/**/*.js');
    return gulp.src(glob, { since: gulp.lastRun('scripts') })
      .pipe(cached('ugly'))
      .pipe(uglify())
      .pipe(remember('ugly'))
      .pipe(concat('main.min.js'))
      .pipe(gulp.dest('dist/scripts'));
  })
);
```

继续使用 since 和 gulp.lastRun，保证每次都只读取新文件

把文件传递给缓存，更新缓存里的文件

最后把所有文件合并起来

uglify 以后，要取回所有处理过的文件；加上之前的文件后，文件流变得更大

这个优化大概可以把整个过程的耗时从 6、7 秒（取决于及其性能），降低到毫秒级别，新的更改眨眼之间就能构建好。

### 6.1.3 处理被删除的文件

优化了代码之后，现在你可以用增量构建更快地迭代了。这些改变全都是基于虚拟文件系统和在内存中保存的文件的。每次调用 cached，都要更新内存，每次调用 remember 都要把文件重新放到文件流里，然后再放到真正的文件系统中。这里存在一个问题：虚拟文件系统并不知道硬盘里的文件发生了什么事。

每次你调用 remember 把文件放到文件流中时，你会把所有文件都放进去，无论它们在真正的文件系统里还存在与否。如果 a.js 文件已经从硬盘里被删除了，但是还存在于内存中的构建缓存中，那在调用 remember 时它就被会放到文件流中。因此，你必须要想办法管理缓存。

你可以通过使用文件 watcher 的事件绑定机制。在第 3 章中，你已经学会了 watcher 是如何工作的，以及如何在文件修改时调用 task。你可以监听这些文件系统中发生的事件，然后执行自己的函数。图 6-7 展示了要做的事情。

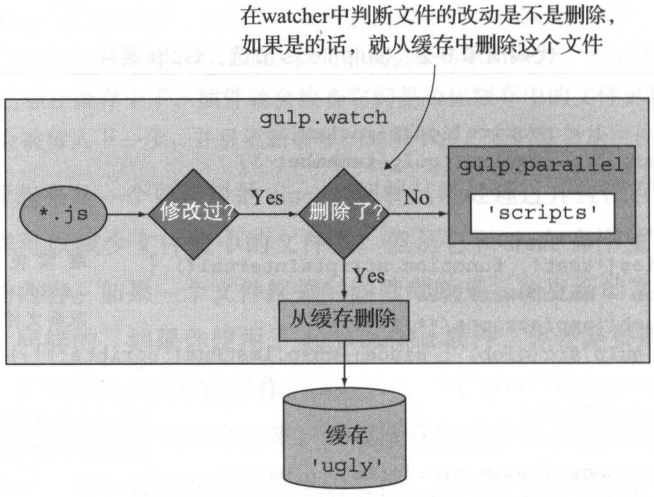



图 6-7 如果一个文件被删除了，它还是会在缓存中。这意味着它又会被 gulp-remember 重新添加到文件流中，即使这个文件已经不存在了。因此，你必须通过 Gulp watcher 监听事件，从缓存中删除已删除的文件

gulp.watch 会返回一个对象，你可以在上面监听事件。下面的代码是具体的实现。

**代码清单 6-3　Gulpfile.js/function watcher()**

```
var slash = require('slash');
var path  = require('path');
var watcher =
  gulp.watch(['app/scripts/**/*.js'], gulp.parallel('scripts'));

watcher.on('unlink', function (filepath) {
  delete cached.caches['ugly'][slash(path.join(__dirname,filepath))];
  remember.forget('ugly', slash(path.join(__dirname,filepath)));
});
```

slash 包是用来管理系统独立文件路径的 ❶

gulp.watch 方法返回一个 watcher 对象 ❷

把这个元素从 gulp-remember 缓存中删除 ❹

watcher 对象会发布不同的事件 ❸

我们要使用 slash 包来确保我们的代码在 *NIX，OSX 和 Windows 下都能运行❶。我们把 watcher 对象复制给 watcher 变量❷。有一个事件叫作 unlink 事件，我们可以监听这些事件并且绑定上回调函数❸。unlink 事件会在一个文件从文件系统中被删除时触发，然后我们把这个文件从 gulp-cached 插件的缓存中删除❹。

这样一来，我们的增量构建就完成了。那些耗时的操作现在只会针对需要处理的文件进行了。本来每次迭代可能要花超过 5 秒的时间，现在只要一秒不到就行了。我们还追踪了文件系统的变化，维护了一个缓存。让我们接着处理下一个 task。

## 6.2　通过 sourcemap 来调试源文件

尽管我们的构建速度现在快得像闪电一样，但是当我们打开浏览器的开发者工具时，我们看到的编译结果依然像天书一样。这对于生产环境很好，因为这意味着我们的代码被优化得很好。但是对于开发而言，我们还是希望看到原始的代码。回顾一下一开始的例子：在浏览器开发者工具中调试生产环境的代码。这根本就是不可能的！我们依赖开发者工具才能知道我们的代码运行得怎么样。我们想要知道变量的值是怎么更改的，错误是怎么产生的。当运行 JavaScript 时，浏览器的开发工具提供了非常复杂的调试环境，和 Java 或者 C# 等语言的调试工具不相上下。它们只有一个不同，那就是 JavaScript 的调试不是在开发环境中进行的（像 C# 的 Visual Studio 和 Java 的 Eclipse），而是在代码的执行环境——浏览器中进行的。浏览器运行的是编译好的、可以部署的代码，与之对应的是其他语言的二进制文件。没有人会想调试二进制文件的。图 6-8 是一个例子。




图 6-8 浏览器开发者工具。在第一个图里，你什么也看不出来。有些浏览器，比如 Chrome，可以在一定程度上格式化代码，至少能够看出一点压缩后的代码的结构。但是你还是看不出来变量名和函数名，就和猜谜一样。为了克服这个问题（像在开发环境一样调试生产环境的代码），浏览器开发商 Google 和 Mozilla 合力推出了 sourcemap。sourcemap 是一种能够追踪编译代码源文件的技术。在本节中，你会了解如何通过 Gulp 生成 sourcemap

### 6.2.1　什么是 sourcemap

sourcemap 能够让浏览器知道如何获取加载的代码的源文件。有两种办法能够让浏览器知道 sourcemap 的存在。一种是在编译好的代码里的注释中加上 sourcemap 的 URL，另一种是直接把 sourcemap 添加在代码里。

sourcemap 的内容是把编译好的代码分割成很多小块，每一小块在 sourcemap 里都有引用，分别包括下面的属性：

- 代码来自于哪个源文件。
- 代码来自于源文件的哪几行。
- 如果代码用到了其他部分的变量，还会包括原来变量的名字。

图 6-9 展示了一个基本的 sourcemap 的一部分。

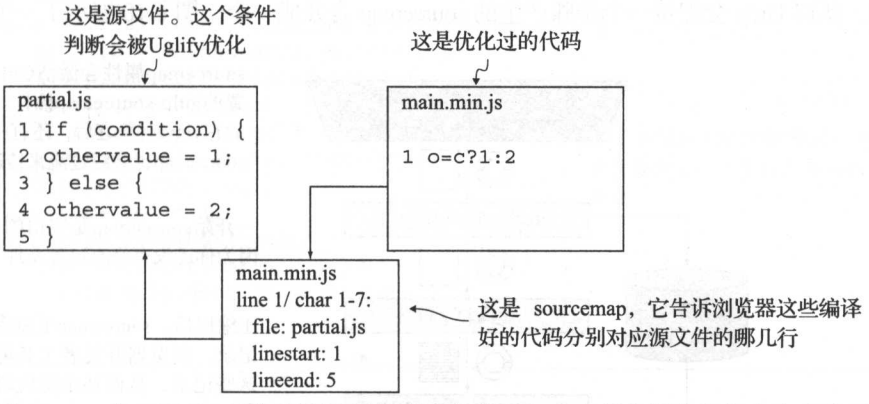



图 6-9　一个 sourcemap 保存了代码的源文件信息。它会保存源文件的名字和代码在源文件里的起始位置

另外，源文件的代码也被包括在 sourcemap 里。这样一来就可以确保任何设备不需要源文件都能加载 sourcemap。

如果一个浏览器，比如 Chrome 或 Firefox 检测到了 sourcemap，它就会加载 sourcemap，并且在开发者工具中显示原始代码。sourcemap 是伴随着对源文件的修改而生成的。比如说 Uglify，它会记录它对文件做的所有修改，然后把相应的信息保存在 sourcemap 里。很多工具和 Gulp 插件都支持生成 sourcemap，这样开发者就可以非常轻松地使用 sourcemap 了。

---



注意　sourcemap 如今已经非常普及了，很多工具和插件都能生成。但是如果一个插件不支持 sourcemap，Gulp 最后的输出结果就会无视它。如果你遇到了这个情

况，那么这个插件可能有问题。你应该试着找个替代品或者更新它。大多数转换文件的插件都支持 sourcemap。那些不支持的插件都已经被列在黑名单里了。

sourcemap 已经成为了一个非常重要的功能，因为很多开发者都经常依赖代码转换编译器，如 CoffeeScript、Typescript 或者 Babel.js。这些工具让开发者可以用一种更现代、更复杂的语言来写 JavaScript，最后再转换成浏览器能运行的 JavaScript。由于最后输出的结果和原来的代码已经不是同一个语言了，所以 sourcemap 在调试的时候就非常重要。CSS 预处理器像 LESS 和 Sass 一样，它们也可以生成 sourcemap。

### 6.2.2 Gulp 内置的 sourcemap

Gulp 内置支持 sourcemap 的功能，而且还有一个特点：由于文件通常都会经过好几个转换步骤，所以 Gulp 会把每一个步骤产生的 sourcemap 合并成一个。图 6-10 展示了一个例子。

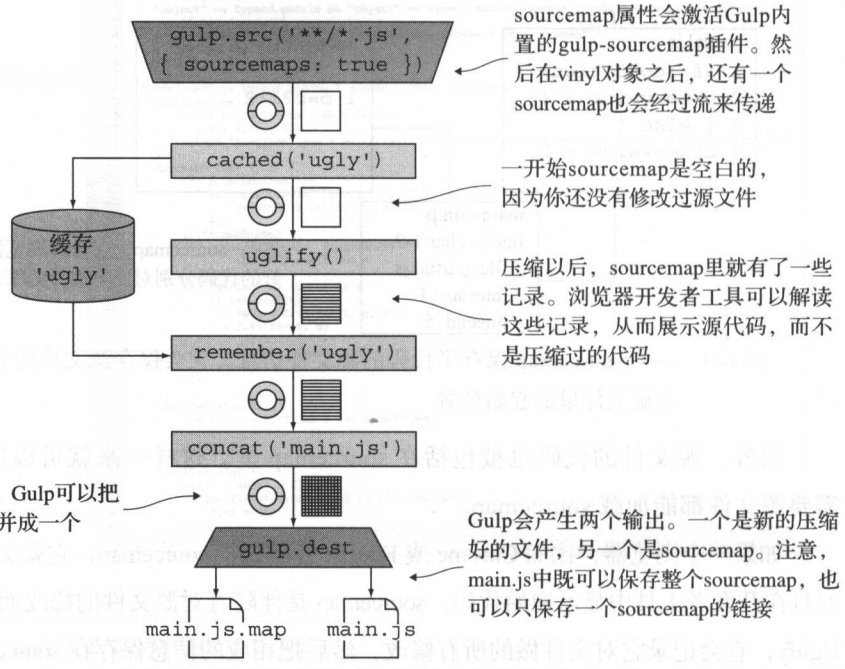



图 6-10 内置的 sourcemap 功能可以给虚拟文件对象新增一个属性叫 sourcemap。sourcemap 里保存的都是你进行的文件转换的记录：哪一行哪一个字符被修改了，原来是什么样子，结果又是什么样子。sourcemap 可以被浏览器解读，从而重现源代码，即使浏览器并没有接触过源代码。Gulp sourcemap 是可以合并的，也就是说每个新 task 都会根据新的数据去修改 sourcemap，所以最后的 sourcemap 还是能够对应到源代码

如果有多个文件经过 Uglify 被压缩，那么每个文件都会生成一个 sourcemap。下一步是合并文件。gulp-concat 也会产生一个 sourcemap，指向上一步的内容。为了能追踪到一开始的文件内容，Gulp 插件可以把之前的 sourcemap 和新的 sourcemap 结合起来。也就是说，新产生的 sourcemap 记录也可以追踪到源文件。


注意　用来合并 sourcemap 的 NPM 包名叫 vinyl-sourcemaps-apply。

这个技术既可以用在 JavaScript 上，也可以用在 CSS 上。下面的代码是添加了 sourcemap 的 scripts task[⊖]。

**代码清单 6-4　Gulpfile.js**

```
gulp.task('scripts',
  gulp.series('test', function scriptsInternal() {
    var glob = mainBowerFiles('**/*.js');
    glob.push('app/scripts/**/*.js');
    return gulp.src(glob, {
        sourcemaps: true,                    <-- 为了 sourcemap 新增的一行，把它设置为 true 来启用 sourcemap 插件
      since: gulp.lastRun('scripts')
    }).pipe(cached('ugly'))
      .pipe(uglify())
      .pipe(remember('ugly'))
      .pipe(concat('main.min.js'))
      .pipe(gulp.dest('dist/scripts'));
  })
);
```

保存之后，sourcemap 就会随着代码一起生成了。另外，由于源代码也会被添加到 sourcemap 中，所以它的体积会很大。这么做是为了确保开发者在浏览器里能看到正确的源代码。但是在生产环境中就没有这个必要了，所以我们要做一些修改。

### 6.2.3　sourcemap 插件

sourcemap 可以保存在编译好的文件中，也可以保存在额外的文件中，只要在编译好的文件里面添加一行注释引用它就可以了。Gulp 内置的 sourcemap 隐藏了很多细节。你可以使用 gulp-sourcemap 插件来配置更多的功能。这个插件也是虚拟文件系统自己使用的插件，所以最后产生的结果不会不同。只是用法会有变化，如图 6-11 所示。

⊖ 作者提到的 Gulp 内置的 sourcemap 是 Gulp 4 测试中的特性，甚至不在文档中，因此不推荐使用。推荐使用 6.2.3 节中使用插件创建 sourcemap 的方式。——译者注

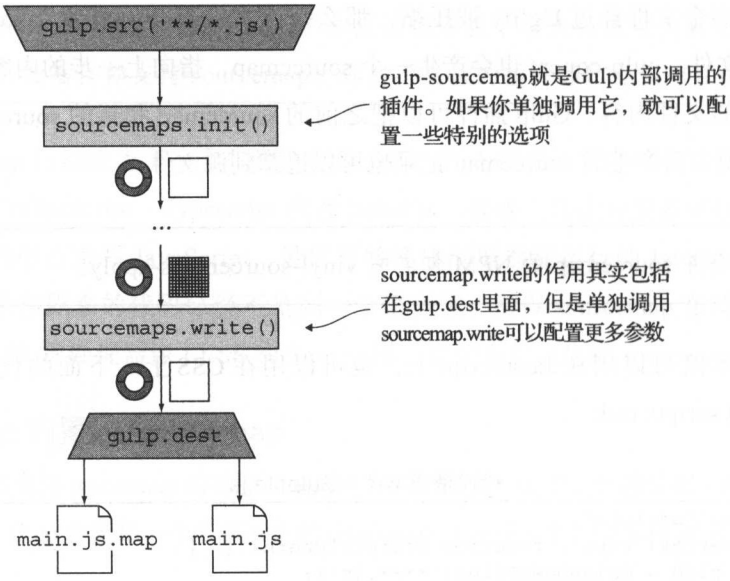



图 6-11 gulp-sourcemap 插件是 Gulp 中的 sourcemap 所用的插件。它内置在 Gulp 中，但是也可以单独使用。当单独使用它的时候，你可以更好地控制 sourcemap 的输出。当你想把 sourcemap 保存在别的文件或者使用其他 Node.js 软件的 sourcemap 时，这会非常有用

你现在是把 gulp-sourcemap 作为流程的一部分使用。相比于使用内置的 sourcemap，它支持更多的选项：

- 你可以指定保存 sourcemap 的目录。如果目录名是空的，就会添加到编译出来的文件中。
- 你可以把源代码从 source 中分离出来。
- 你可以定义根目录（source root）。默认情况下，它就叫 source。如果同时生成了 JavaScript 和 CSS 的代码，那么所有文件都会展示在这个目录里。你可以给这个目录起个别的名字，比如 js-source 和 css-source 来加以区分。

还有一些其他选项，不过上面这些就是你最常用的了。要了解更多选项，可以去阅读文档。用 gulp-sourcemap 替代内置 sourcemap 的代码如下。

**代码清单 6-5 Gulpfile.js**

```
var sourcemaps = require('gulp-sourcemaps');

gulp.task('scripts',
  gulp.series('test', function scriptsInternal() {
    var glob = mainBowerFiles('**/*.js');
```

```
    glob.push('app/scripts/**/*.js');
    return gulp.src(glob)
      .pipe(sourcemaps.init())
      .pipe(cached('ugly'))
      .pipe(uglify())
      .pipe(remember('ugly'))
      .pipe(concat('main.min.js'))
      .pipe(sourcemaps.write('.', {
        sourceRoot: 'js-source'
      }))
      .pipe(gulp.dest('dist/scripts'));
  })
);
```

在构建管道中调用插件来启用 sourcemap

❶ 转换完文件后，再次调用 sourcemap 插件，但是这次调用的是 write 方法

第二个参数是一个
❷ 配置对象

这里我们传递了两个参数，第一个是要保存 sourcemap 的目录❶。一个点表示和文件保存在同一个目录下。在第二个参数里，我们把 source root 的重命名为 js-source，编译在浏览器开发者工具中找到它❷。

现在我们已经完成了一开始提到的第二个任务。现在你已经可以在浏览器里看到源代码，进行调试了。但是到目前为止，我们都还是在构建开发环境的代码，生产环境怎么办呢？下一章你就知道了。

## 6.3　切换环境

现在我们已经优化了本地开发环境，让它用起来更舒服了，但是我们目前忽视了生产环境。尽管我们现在输出的代码已经高度优化过并且可以部署了，但是要生成真正的生产环境代码，我们的工作流还有一些不足：

- 生产环境不需要 sourcemap。这个功能是本地开发环境才需要的。对于生产环境，它太大了。而且，你一定也不想让你的代码被人随意读懂。
- 生产环境不需要运行本地开发服务器。
- 生产环境不需要迭代，也就不需要 watcher，只要构建一次就可以了。

在本节中，我们始终遵守最终一个原则：不要复制构建管道。你一定也不想为了两套环境而写很多冗余的代码，如图 6-12 所示。

要实现我们的目标，就要修改 task，让它更加智能。我们应该在不修改现有的构建管道的前提下，把 sourcemap 插件变成可选的一部分。这一节就会告诉你要怎么办。

### 6.3.1　noop()

要停止构建管道中某个步骤的执行，你可以什么都不执行。这听起来有点无厘头，

但是请仔细想想：假如你有一个开关，可以把流的内容导向不同方向，你在每一个步骤都可以切换这个开关，然后控制流传递到下一个步骤。图 6-13 形象地展示了这个过程。

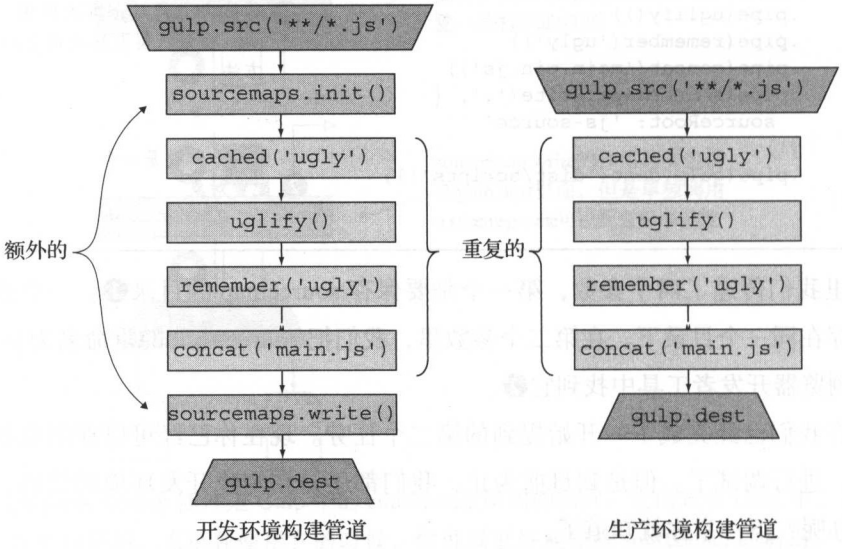



图 6-12 生产环境的构建管道和开发环境的构建管道的差异。唯一会对最后结果产生影响的步骤是 sourcemap。额外的 sourcemap 应该是一个可选项，只有开发环境才会用到。剩下的部分都是一样的

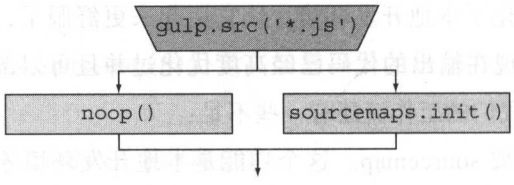



图 6-13 管道开关的示意图。根据条件，你可以把流导向一个任务或者另一个任务。另一个任务什么也不会做，只是把内容传递给下一个步骤。

要么执行 task，要么什么都不做，把内容传给下一个步骤。这种任务就叫作 noop。要创建它，你可以使用 NPM 提供的一个包，through2 包就非常合适。详情请看下面的代码。

**代码清单 6-6 Gulpfile.js**

```
var through = require('through2');

var noop = function() {
    return through.obj();
};
```

through2 就是我们所选的包，用来创建管道中的步骤

through.obj 为流创建一个新的步骤 ❶

通常这个函数会接受一个转换函数作为参数，但是如果你什么都不传，那么到时

候内容就会直接从流中穿过❶。obj 表示你要传递的是对象，Vinyl 文件对象也是对象。

noop 函数用起来就和 Gulp 插件一样，区别在于它不会修改任何文件内容，只是把内容传递给下一个步骤。有了这个函数，接下来我们就要创建开关了。

### 6.3.2　dev() 和 prod() 选择函数

所谓的开关实际上可以转换为两个函数：dev() 和 prod()。前者用来执行开发环境的任务，后者用来执行生产环境的任务。它们接受一个参数，就是满足条件时要执行的 task，等到条件真的满足时，就会执行它。图 6-14 展示了它们是如何工作的。

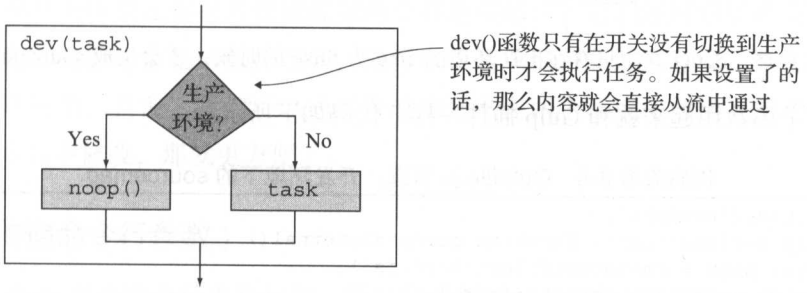



图 6-14　dev() 函数会检查开关是否切换到了生产环境。如果是的话，那它就会把内容传给 noop 函数，什么都不会做。如果不是的话，那么 dev() 就会执行传入的 task

控制切换的条件保存在 Gulpfile 的一个布尔值变量中，把 isprod flag 设置成 false 就表示开发环境。下面的代码是具体实现。

**代码清单 6-7　Gulpfile.js**

```
如果 production 的 flag 被设置成 false
的话，就会执行开发环境的任务，反之亦然
    var isprod = false;

    var dev = function(task) {
        return isprod ? noop() : task;
    };

    var prod = function(task) {
        return isprod ? task : noop();
    };
```

dev 接受一个参数：在开发环境中要执行的任务

如果这个 flag 被设置成 true（表示生产环境），就会把文件传给 noop()；否则就会把这个任务添加到构建管道中

prod() 和 dev() 相反，但是做的事情类似

这两个函数会返回一个流的函数，通过它就可以传递对象流了。你可以在 Gulp 中调用它们。如图 6-15 所示。

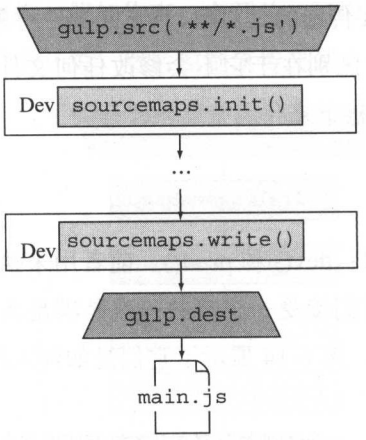



图 6-15 dev() 方法只有在生产环境的 flag 为 false 的时候，才会生成 sourcemap

这两个函数用起来就和 Gulp 插件一样。代码如下所示。

**代码清单 6-8 Gulpfile.js 节选：开发环境下的 sourcemap**

```
gulp.task('scripts',
  gulp.series('test', function scriptsInternal() {
    var glob = mainBowerFiles('**/*.js');
    glob.push('app/scripts/**/*.js');
    return gulp.src(glob)
      .pipe(dev(sourcemaps.init()))
      .pipe(cached('ugly'))
      .pipe(uglify())
      .pipe(remember('ugly'))
      .pipe(concat('main.min.js'))
      .pipe(dev(sourcemaps.write('.', {
      sourceRoot: 'js-source'
      })))
      .pipe(gulp.dest('dist/scripts'));
  })
);
```

dev 函数确保只有在开发环境中才会生成 sourcemap

如此一来，两个环境的构建流程就都准备好了。你可以使用同一个开关来控制 Browsersync 和 watcher，如下代码所示。

**代码清单 6-9 Gulpfile.js 节选：生产环境开关**

```
gulp.task('server', function(done) {
  if(!isprod) {
    bSync({
      server: { baseDir: ['dist', 'app'] }
    })
  }
  done();
});
```

这个 if 判断会在生产环境中禁用开发环境中的功能

default task 中的 watcher 也可以这么改，请看下面的代码：

**代码清单 6-10　Gulpfile.js 函数 watcher**

```
function watcher(done) {
  if(!isprod) {
    var watcher = gulp.watch([...], gulp.parallel('scripts'));

    // …

    gulp.watch('app/styles/**/*.less', gulp.parallel('styles'));
    gulp.watch('dist/**/*', bSync.reload);
  }
})
```

这两个开关可以让你很方便地控制两个环境的输出。它们还可以扩展支持更多的环境。因为开关实际上是函数控制的，你可以很简单地判断开发环境、预生产环境或者其他生产环境。目前，我们可以通过修改变量来在两个环境中切换。但是如果能够直接在命令行中修改，那就更方便了。

### 6.3.3　传递命令行参数

由于 Gulp 是在命令行中执行的，所以你不仅可以在命令行中执行 task，还可以传递参数。这样你就可以根据自定义 flag 来扩展它的功能。Node.js 有一个很方便的插件叫 yargs，它是用来解析参数的。我们的目标是添加一个 flag，用来表示生产环境，用起来应该是这个样子：

```
$ gulp --env=prod
```

前面有两个连字符的参数会自动被解析成 flag，它们可以根据名字来访问，代码如下：

**代码清单 6-11　Gulpfile.js 节选**

```
var arguments = require('yargs').argv;
var isprod = (arguments.env === 'prod');
```

引入 yargs 包，保存在 arguments 变量中

当你添加一个两个连字符的参数的时候，arguments 对象流里就会有一个同名的属性 ❶

这一行❶引入了传递的参数。我们检查它是不是 prod，是的话就代表生产环境，否则就是开发环境。

搞定了！现在你有了一个切换不同环境的非常方便的方法，你还可以在命令行中

控制其他 Gulpfile 内部的变量（比如输出目录）：

```
$ gulp --outputFolder=dist
```

Gulpfile 里这样写：

```
.pipe(gulp.dest(arguments.outputFolder + '/scripts/'))
```

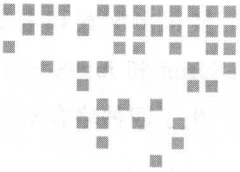
现在尽情地把你的代码部署到服务器上吧。

## 6.4 总结

在本章中，你学会了如何为开发和生产环境构建代码：

- 你知道了什么是增量构建，以及怎么在 Gulp 实现它。你还创建了构建缓存来加速迭代，这样就不用每次都重新编译所有代码了。
- sourcemap 帮助你在浏览器的开发工具中阅读源代码。你既可以用 Gulp 内置的 sourcemap，也可以用插件来更灵活地配置输出结果。
- 最后，你创建了开关来切换开发和生产环境的构建流程。为了实现它，你写了第一个插件：一个用来把虚拟文件对象传递给下一个步骤的插件。
- 你还学会了如何使用 yargs 包来解析命令行参数，这样你就不需要冗余的代码来控制开关了。

我们对流的优化才刚刚开始。下一章，我们会学习更多的流的技术，来改进 Gulpfile。



第 7 章 Chapter 7

# 使 用 流

**本章包括：**

- 用合并流和穿插流来结合不同的输入。
- 使用流数组处理重复的流。
- 创建流的 snippet。
- 使用流队列处理流的顺序。
- 动态地筛选流的内容。

之前我们已经为不同的代码创建了不同的构建管道，比如 CSS 和 JavaScript。样式和脚本有着根本性的不同，所以我们要区别对待。这就导致了这两类代码分别有一个构建管道，正如你在第 2 章所见。但是还有另一种情况：我们要处理同一类文件，过程相似，但是有细微的不同，以致于我们要复制原有的构建管道代码。遇到这种情况怎么办呢？

回想第 2 章一开始的 scirpt task。这个 task 构建出了一个 JavaScript 代码包，所有的资源都被整合进了一个文件。假如我们要创建多个文件呢？更麻烦的是，如果我们要把不同的 JavaScript 预处理语言写的代码编译成一个文件又该怎么办？

CoffeeScript 和 TypeScript 都要首先被编译成普通的 JavaScript，才能合并和压缩这两部分代码。

实话实说，任何构建工具都能实现这个过程。但是要能够整合各种形式、各种语

言的输入，需要很多配置和重复的代码。比如说，看一下图 7-1。我们有两种类型的输入（CoffeeScript 和 JavaScript）和两种类型的输出（第三方代码和我们自己应用的代码），但是却产生了四种略有不同的构建管道。

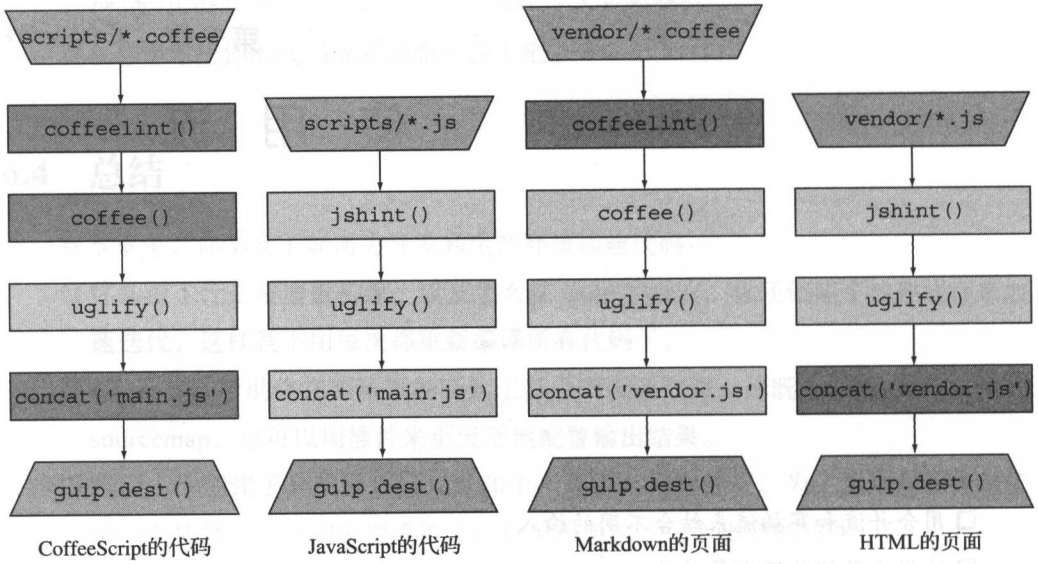



图 7-1 本节一开始提出的问题。你想编译不同的 JavaScript 代码。有些是用 CoffeeScript 写的，有些是用 JavaScript 写的，有些两者兼有。构建过程大致是一样的，但是输入内容有着显著差异。我们不想一次又一次地复制同样的构建管道，我们希望创建一个构建管道就能处理所有的输出和配置

借助 Gulp，我们能做得更好。由于 Gulp 是基于 Node.js 流的，你可以充分利用这个技术的特性和生态，来创造出灵活的可以适合任何配置的构建管道，而不用一遍遍地复制代码和配置。写 Gulpfile 的一个原则就是：能写代码就不要写配置。接下来你就会看到具体应该怎么办。

在本章中，我们会调整和重构第 2 章中的构建管道，以适应各种各样的输入，同时还能保持它原有的可维护性。

读完本章，你就可以变成一个专家级的管道工了。

**本章示例** 本章的所有例子基于的都不是我们原来的项目，而是一个经过修改的独立的项目。如果你想下载这些项目，请访问 https://github.com/frontend-tooling/chapter-7-examples。

## 7.1 处理不同类型的输入

截至目前，我们的构建管道都是笔直的。先选择一堆文件，然后把它们传递给一系列处理步骤，最后把输出结果保存在文件系统中的某个地方。图 7-2 为第 2 章中的 scripts task。

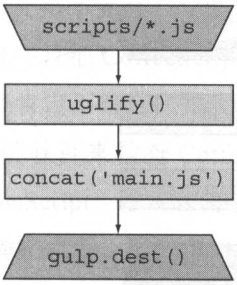



图 7-2 原始的 scirpts 构建管道。先是选中所有的 JavaScript 文件，然后用 Uglify 压缩，最后把它们合并成一个文件。接下来我们就要复用这个管道，并且让它兼容各种输入

如果简单地描述这个管道的话，它只有一个主要的流程。它专门处理一种类型的输入：以 .js 结尾的文件。如果你有别的类型的输入，也需要走这么一套类似的流程，但是在事前又需要做一些其他的处理，那么你可能就得在这个管道的基础上进行修改，然后再创建一个新的 Gulp task 和流了。然而一个优秀的软件开发者是绝对不会复制代码的，所以让我们看看有没有什么办法能够利用流的力量来解决这个问题。

### 7.1.1 穿插流

对于第一个场景，也就是把 CoffeeScript 和 JavaScript 打成一个包的需求。实际情况可能是，一个项目一开始处于某种架构的考虑，使用了 CoffeeScript。结果后来想用 JavaScript 来写其中一些新组件。要做到这件事，并不需要把所有 CoffeeScript 转换成 JavaScript，只要修改构件流程，把 CoffeeScript 和 JavaScript 编译到一个包里就可以了。

有一个办法可以实现这个目标，就是使用穿插流（passthrough stream）。我们用 Gulp 的时候，通常用可读流在一开始选择文件，使用可写流把流传给别的 task，或者在最后把内容保存起来。穿插流既可读，也可写。gulp.src 既可以创建可读流，也可以创建可写流。在构建流程中的任何一步，你都可以往流里添加（或者传递）新文件。这

样这些文件会跳过之前的处理步骤，直接传给下一个 task，如图 7-3 所示。

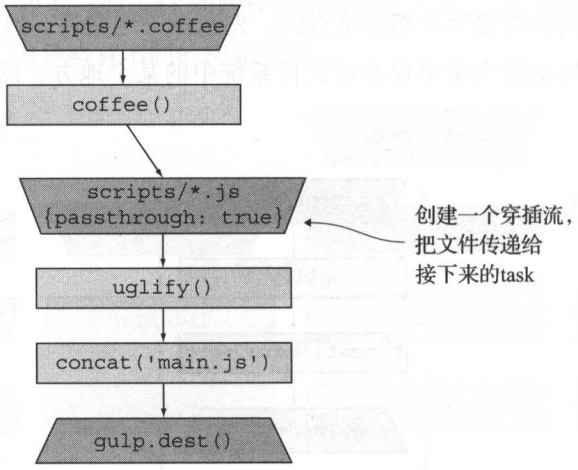



图 7-3 使用穿插流可以在流的中途选择其他的文件，然后把它们添加到流中。这样一来就可以合并不同类型的输入，避免有些步骤因此出错。比如 coffee() 在处理普通的 JavaScript 时就会出错

只要给 gulp.src 添加一个参数，就能做到上面这件事。这个参数的作用是告诉 gulp.src 要接受之前的流传递来的内容。下面就是例子代码：

**代码清单 7-1 Gulpfile.js 片段**

```
gulp.task('scripts', function() {
  return gulp.src('src/scripts/**/*.coffee')
    .pipe(coffee())
    .pipe(gulp.src('src/scripts/**/*.js', {passthrough: true}))
    .pipe(concat('main.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist/scripts'));
});
```

把之前的结果传递给另一个 gulp.src ❷

选中所有的 CoffeeScript 文件，然后传递给 coffee ❶

我们先处理的不是 JavaScript 文件，而是 CoffeeScript 文件❶。通常，❷这样做会覆盖前面的流，并且只会选中 .js 文件，但是加了 passthrough 之后，就能够把之前步骤的结果添加到新的流里。在这个例子中，你把新文件添加到哪里，新文件就会从哪里开始继续传递。

通过使用穿插流，就可以把不同的输入合成一个流，用同样的步骤来处理它们了。我们把逻辑都放在一个流里面，避免了很多配置和冗余的代码。如果你的管道中有一个步骤变了，也只需要修改一个地方就行了。如果又有另一种输入要加进来，也可以

用同样的流来处理。

### 7.1.2 合并流

穿插流在上个场景中工作得很好，但是看起来似乎少了点什么。无法单独处理后合并到流里的那部分文件。设想一下，你要给 CoffeeScript 和 JavaScript 分别添加代码风格检查的步骤。由于 CoffeeScript 是个完全不同的语言（更像是 Ruby），所以就不能使用 JavaScript 的检查工具。而且一旦把 CoffeeScript 编译好了，编译出来的代码也无法通过检查，因为编译出来的代码肯定不符合你的代码要求。

你可以用一个叫 CoffeeLint 的软件来检查 CoffeeScript。但是这个工具只能检查 CoffeeScript，不能检查普通的 JavaScript。你现在碰到的问题是：你有两个流，要根据输入的类型分别处理，等它们处理完成后，再把它们合并到一个流中。

你可以使用合并流（merge stream）来做这件事。合并流可以并行处理两个单独的流，然后把它们的结果合并到一个流，供接下来的 task 处理。这个并行的流可以处理各种步骤。

让我们用合并流代替穿插流，重写 7.1.1 节的例子，详情请看图 7-4。

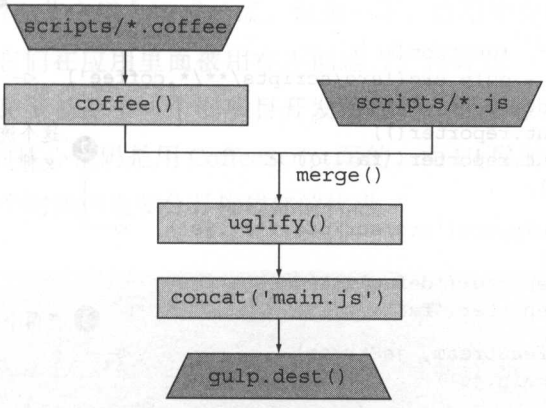



图 7-4 这个管道用了合并流，但是做的事情和穿插流是一样的

图 7-4 展示的过程和图 7-3 是一样的。Gulp 的穿插流是通过一个合并流的包来实现的，你只是在重写一个同样的功能，但是却更加容易扩展。有了两个独立的流，你就可以分别给它们想加多少 task 就加多少 task，如图 7-5 所示。

为了实现这一点，我们要使用一个很流行的包：merge2，如下面的代码所示。它可以用来传递 Gulp 创建的流。

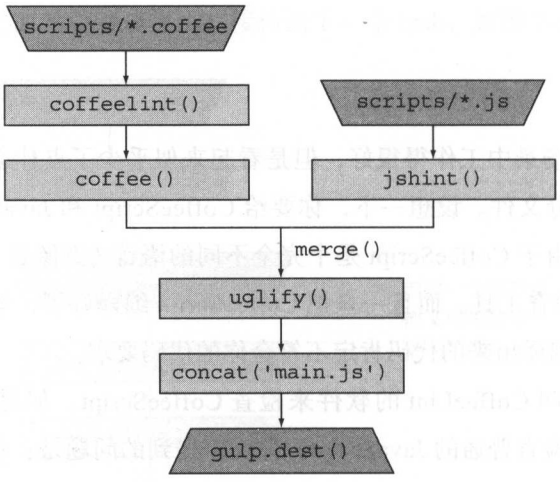



图 7-5 通过合并流，你可以在它们合并之前，分别对不同的输入执行不同的 task，等到合并后，再执行一些共同的 task

**代码清单 7-2 Gulpfile.js 片段**

```
var coffeelint = require('gulp-coffeelint');
var coffee     = require('gulp-coffee');
var merge      = require('merge2');

...

gulp.task('scripts', function() {
  var coffeeStream = gulp.src('src/scripts/**/*.coffee')
    .pipe(coffeelint())
    .pipe(coffeelint.reporter())
    .pipe(coffeelint.reporter('fail'))
    .pipe(coffee());

  var jsStream = gulp.src('src/scripts/**/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
    .pipe(jshint.reporter('fail'));

  return merge(coffeeStream, jsStream)
    .pipe(concat('main.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist/scripts'));
});
```

❶ 这个流选择 CoffeeScript 文件

❷ 这个流选中 JavaScript 文件，并且执行 JSHint

❸ 把两个流合并成一个

❹ 把它们传递给共同的 task：压缩和合并

我们照常选择文件，照常运行 CoffeeScript 插件和编译 CoffeeScript ❶。我们像第 2 章里讨论过的那样处理 JSHint 的 reporter，第 2 章里已经讲过这些代码的含义了❷。最关键的是这一步❸。这一步以后，我们就有了一个既包括编译好的 CoffeeScript 文件也包括 JavaScript 文件的流了。然后把它们传给接下来的步骤❹。

尽管穿插流也提供了很棒的方法来往流里添加文件，但是如果你要分别执行更多、更复杂的 task，那么合并流是更好的选择。所以，当你仅仅只是要添加新文件时，不妨使用穿插流；当你的需求更复杂时，选择合并流。

## 7.2 处理不同的输出

通过穿插流和合并流，你现在已经可以把多个不同输入合并成一个输出了。这意味着不管你有多少个不同类型的输入，最后总是能得到一个结果。这主要是因为你的管道中有一个特别的 task——gulp-concat，正是它把所有的文件都打包成了一个。

对于之前的场景，这已经完全足够了。但是你想不想更进一步，试着用构建代码创建多个 JavaScript 包？比如，你想把输出的代码分成第三方代码和你自己的代码。如何复用之前的流，同时又确保输入和输出都能够分开处理，而不会影响其他的构建流程呢？这一节就会告诉你该怎么做。

### 7.2.1 在 task 中根据参数创建流

让我们来复用 7.1 节中组合而成的流。设想一下，应用中有好几个组件。你想分别构建它们，因为它们在应用里面被用在不同地方。再设想一下，你有一个已经开发了很久的项目，甚至是基于一个旧项目开发而成的，而你想复用其中的一部分代码。又比如，应用的核心代码是用 CoffeeScript 写的，而 UI 层的代码是用下个版本的 JavaScript 写的。这个时候你也要分开构建你的代码。

**下一个版本的 JavaScript**

JavaScript 的基础 ECMAScript 标准是在很多年前制订的。TC39 委员会决定哪些语言特性已经发展得足够成熟可以当作标准来实现了（在浏览器和 Node.js 平台中）。由于标准的发展一向是超前于浏览器的，开发者必须要用编译器来把用最新 ECMAScipt 标准（ES2015、ES2016 等，根据年份命名）写的代码编译成浏览器能理解的代码。这些编译器可以把语法糖转换成可执行的代码或者 polyfill（提供浏览器可以直接运行的代码），而这些特性本来是无法运行的。

这样一来，开发者不需要等待浏览器支持，就可以使用语言的最新特性了。由

于浏览器和操作系统的分裂，开发者无法确保自己的应用能够运行在兼容所有特性的浏览器中。

你的应用的两个部分（核心代码和 UI 层）的处理步骤都是一样的：

1）检查代码风格。

2）编译代码。

3）合并代码。

4）压缩代码。

最后两步和之前做的没有区别，但是代码风格检查和编译代码则不然。这两步的差别取决于你使用的技术，有时候它们甚至需要不同的 Gulp 插件才能运行。

借助 Gulp，我们可以非常自然地复用构建流程，即使流程的内容有所改变。回想本书第 1 章所说的 Gulp 作为构建系统的优点。Gulp 所用的就是 JavaScript，没有字符串，也没有会产生阻塞的 API。当然了，我们在定义 task 和使用流的时候还是用了一些 Gulp 的 API，但是剩下的代码完全是我们自己添加的。

记住这一点以后，你就可以用占位符（placeholder）而不是 task 来重构管道，把它变成一个可以用参数配置的流。用给函数传递参数的方式，把占位符替换掉，参见图 7-6。

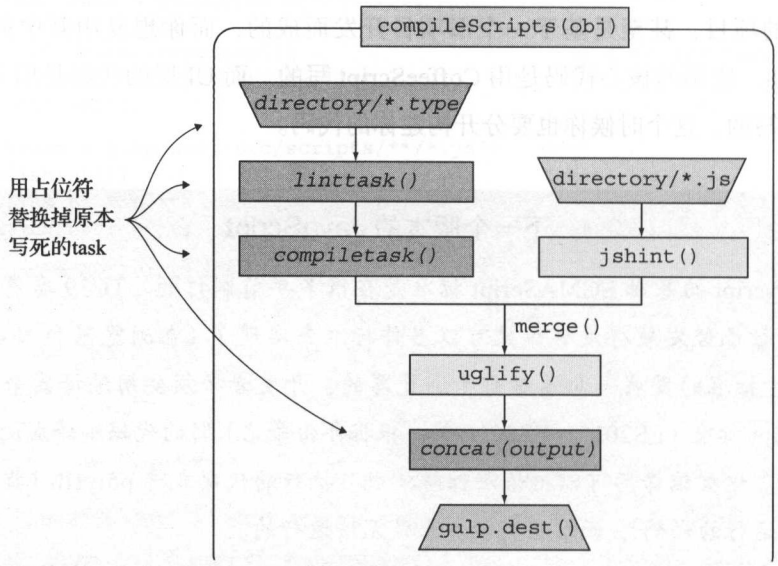



图 7-6 通过用占位符（图中用斜体表示）来替换掉写死的 task，就可以在各个流中复用构建管道。如果把它们重构成函数，就可以在多个流甚至 task 中使用了

这个函数可以被任何 task 调用。下面的实现只包含了两个 task，一个是核心代码的，一个是 UI 层的。两个 task 用的都是不同的编译器。详情请看代码清单 7-3。

**Github 上的示例** ECMAScript 6 的示例需要很多额外的插件，如果都写出来就太多了。如果你想知道更多内容，可以查看 https://github.com/frontend-tooling/chapter-7-examples 里的代码和 package.json 文件。

**代码清单 7-3 Gulpfile.js（节选）**

```
/** new plugins **/
var eslint = require('gulp-eslint');
var babel  = require('gulp-babel');

function compileScripts(param) {
  var transpileStream = gulp.src(param.directory + '**/*.' + param.type)
    .pipe(param.linttask())
    .pipe(param.fail)
    .pipe(param.compiletask());
  var jsStream = gulp.src(param.directory + '**/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('fail'));

  return merge(transpileStream, jsStream)
    .pipe(concat(param.bundle))
    .pipe(uglify())
    .pipe(gulp.dest('dist'));
}

gulp.task('core', function() {
  return compileScripts({
    linttask: coffeelint,
    fail: coffeelint.reporter('fail'),
    compiletask: coffee,
    directory: 'core/',
    type: 'coffee',
    bundle: 'core.js'
  });
});
gulp.task('ui', function() {
  return compileScripts({
    linttask: eslint,
    fail: eslint.failAfterError(),
    compiletask: babel,
    directory: 'ui/',
    type: 'es',
    bundle: 'ui.js'
  });
});
```

compileScripts 是上一个 task 定义的流 ❶

动态地构建 CoffeeScript 的构建流

用 Babel 构建 ES2015/ES2016

可以看到原本我们编译文件的代码，现在被替换成了传递给函数的参数❶。这样

最大的好处是可以指定构建过程，而且只要在一个地方定义好构建流程，就可以在其他地方通过简单地修改参数来替换了。

### 7.2.2 流数组

上一节我们用通过配置流的参数，来一次次地复用同一个流。这一节里让我们把这个想法继续发扬光大。尽管 7.2.1 节的代码工作得很不错，但是它有一个缺点：它需要你每次手动地配置参数对象，才能把流实例化。而且，它把每个流都放在一个独立的 task 中。

如果我们只用一个 scripts task 来执行所有的代码的流，会不会更加方便呢？只用一个 task 就能根据你的配置，来实例化所有的流。

有一个办法可以实现这个目标，那就是流数组（stream array）。流数组，顾名思义，就是由不同的流构成的数组，一次性全部执行。它们充分利用了标准的 JavaScript 数组对象的方法，尤其是 map 方法。

map 方法是用来把一个数组转换成另一个数组的方法。它会迭代处理数组的每个元素，并且对它们执行某种 mapping 函数，变成另一个元素。最后的结果会按原来的顺序保存在一个新数组里。详情请看下面的代码。

**代码清单 7-4 map 方法替换**

```
Array.prototype.map = function(mapFunction) {
  var result = [];
  for(var i = 0; i < this.length;i++) {
    result.push(mapFunction(this[i]));
  }
  return result;
}

[1, 2, 3].map(function(el) {
  return el * 2;
});
```

❶ 这个方法通常已经由浏览器或者 Node 内置了

❷ map 方法会迭代处理数组中的所有元素

❸ 这是最后返回的结果

❹ 把一组数字 map 成它们各自的两倍

如果 map 方法❶还没有实现，那可以通过这段代码来实现。map 是一个数组的方法，它接受一个参数：对每个元素执行的 mapping 函数。map 会对数组的每个元素执行 mapping 函数，并且把结果保存在一个新数组里❷。最后你会得到一个大小和原来一样的数组，其中是根据原数据转换而来的数据❸。最后的结果是 [2, 4, 6] ❹。

当你要批量转换元素时，Array.prototype.map 非常方便。仔细想想，我们可以用它

把一组配置对象转换成一组流。这正是所谓的流数组，参见图 7-7。

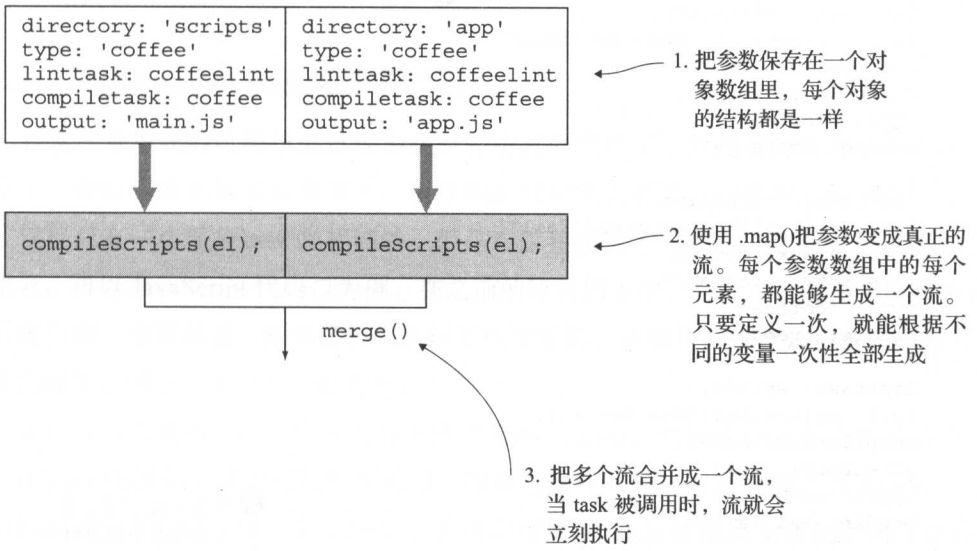



图 7-7 使用 Array.map 函数，你可以用一组配置对象，创建大量的流。在最后使用 merge 方法，把所有的流合并成一个流，然后返回给 task

从一组配置对象，你可以获得一组流。使用之前调到的 merge，你可以把它们合并成一个流来执行。图 7-7 展示了两个变量被 compileScripts 转换成流的过程。由于你的变量现在是数组的形式，所以你可以往里面添加任意数量的元素，如图 7-8 所示。

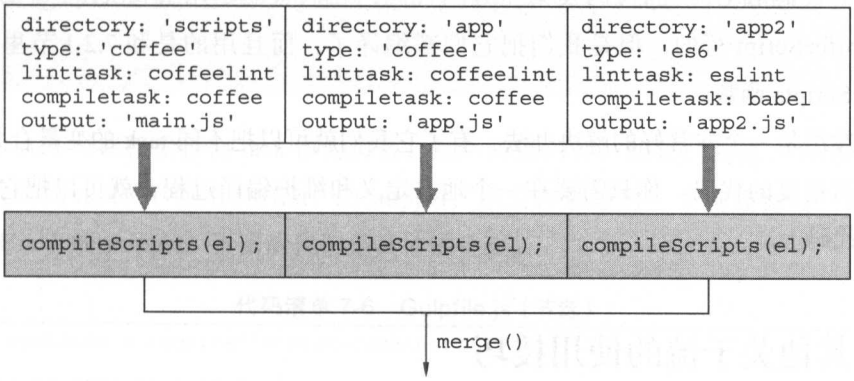



图 7-8 规则一样，但是 task 不一样。现在你可以把多个 CoffeeScript 代码包和 ES6 的代码一起编译了。过程和之前还是一样的，而且你仍然在一个地方就能修改它们

代码清单 7-5 展示了如何在 Gulp 中实现上述流程。

**代码清单 7-5 Gulpfile.js（节选）**

```
var variations = [{                                    ❶ 变量数组
  linttask: coffeelint,
  fail: coffeelint.reporter('fail'),
  compiletask: coffee,
  directory: 'core/',
  type: 'coffee',
  bundle: 'core.js'
},{
  linttask: coffeelint,
  fail: coffeelint.reporter('fail'),
  compiletask: coffee,
  directory: 'vendor/',
  type: 'coffee',
  bundle: 'vendor.js'
},{
  linttask: eslint,
  fail: eslint.failAfterError(),
  compiletask: babel,
  directory: 'ui/',
  type: 'es',
  bundle: 'ui.js'
}];
gulp.task('scripts', function() {
  var streams = variations.map(function(el) {
     return compileScripts(el);
  });
  return merge(streams);
});
```

❷ 借助 map 方法，你可以为每个配置对象创建一个流

当你获得所有的流时，通过 Gulp 把它们合并成一个

你可以想往 variations 数组里添加多少配置对象就添加多少❶。在这个例子中，我们有核心代码和 UI 层的代码及第三方的代码，它们有的是用 JavaScript 写的，有的是用 CoffeeScript 写的，但是我们把它们都编译了，而且用的是和 7.2.1 节里一样的 compileScripts 函数。

流数组是一个非常好的解决办法，有了它我们就可以把不同 task 的变量合并起来，而不用写重复的代码。你只需要在一个地方定义和维护编译过程，就可以把它应用在所有的代码上。

## 7.3 其他关于流的使用技巧

在前两节中，你已经学会了如何把不同格式的输入打包成一个，并如何根据不同的输入变量使用同一个流来创建多个代码包。借助合并流、穿插流和流数组，你已经可以处理大部分复用代码以及创建复杂的流的情况了。但是尽管你可以处理大部分问

题，有时候你可能还是会碰到一些小问题。这一节的目的就是教授大部分你可能会在Gulp中用到的常用的流的使用技巧。

### 7.3.1 使用snippet来避免重复代码

在这一章里我们已经讨论过很多次避免重复的代码了。但是有时候之前的办法都用不了。有时流的变量实在太多了，很难用配置对象去管理；或者有时两个流完全不同，仅仅只有一小部分task是相同的。但是在这些情况下，我们仍然有办法来避免代码重复。再以JavaScript代码包为例，在之前的所有例子中，合并步骤都是在压缩步骤之后进行的。也就是说，除非已经先执行了压缩步骤，否则合并步骤永远都不会执行。让我们确保这两个task总是一起发生。

也许你可以使用一个二合一的task或者流的snippet，然后执行多个task而不是一个。snippet用起来应该和其他插件类似，但是它执行的是一系列插件，而不是一个定义好的函数。要创建这样一个snippet，你可以使用Node.js流的生态系统中的一个包：stream-combiner2。

> **streams2**
>
> 现在你可能已经注意到了，我们所使用的所有和流相关的包最后都有一个2：merge2、through2和stream-combiner2。尽管流的函数是集成在Node.js核心里的，但是这些包却是社区提供的。它们就像是原生的流的API的延续。它们比原生的API更加易用。

这个API可以接受任何task作为参数，然后传递内容。让我们创建一个用来处理第三方代码的task（不使用JSHint）的snippet，然后再创建另一个构建你自己的代码（使用JSHint）的snippet。下面的代码是具体的实现。

**代码清单7-6 Gulpfile.js（节选）**

```
var combiner = require('stream-combiner2');

function combine(output) {
  return combiner.obj(
    uglify(),
    concat(output)
  );
}
```

❶ 调用stream-combiner

```
gulp.task('vendor', function() {
  return gulp.src('vendor/**/*.js')
    .pipe(combine('vendor.js'))
    .pipe(gulp.dest('dist'));
});

gulp.task('scripts', function() {
  return gulp.src('src/**/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('fail'))
    .pipe(combine('bundle.js'))
    .pipe(gulp.dest('dist'));
});
```

❷ 像使用插件一样使用之前定义好的 snippet

我们用 stream-combiner 的 object 模式来调用它（因为 Gulp 流的内容就是对象），在其中执行你所需的 task ❶。你不需要自己在 task 之间传递数据，stream-combiner 会确保数据的传递的❷。

如果你想往管道里添加其他步骤，你可以在 snippet 里添加，这样就可以在所有用到 snippet 的 task 里生效了。

### 7.3.2 使用流队列来管理顺序

我们合并流的时候，用的是 merge2。这个包之所以特别有用，是因为它能保持 Gulp 的异步，而且同时并行地执行所有的流。但是有时候你并不希望这样。设想一个处理多个来源的 CSS 的场景：

- 一个文件来自 Bower 组件。
- 有一些复用自其他项目的库。这些代码使用了 CSS 的 @import 规则，引用了更多的文件。
- 最后，有些新代码使用 LESS 编写。

在 CSS 中，确保顺序是必要的。“层叠”（CSS 全名叫作层叠样式表）的特性使得后面的声明会覆盖掉前面的规则。然而 merge2 有可能会把顺序搞乱，因为有些任务耗时要比其他的长。这个时候你就可以用流队列了，流队列可以确保流的输出结果是完全按照队列顺序产生的。看看下面的代码，里面使用了 streamqueue 包。

**代码清单 7-7 Gulpfile.js（节选）**

```
var queue      = require('streamqueue').obj;
var cssimport  = require('gulp-cssimport');
```

❶ 直接使用 streamqueue 的 obj 方法

```
gulp.task('styles', function() {
  return queue(gulp.src(mainBowerFiles('*.css')),
    gulp.src('lib/lib.css')
      .pipe(cssimport()),
    gulp.src('styles/main.less')
      .pipe(less())
  ).pipe(autoprefixer())
  .pipe(concat('main.css'))
  .pipe(gulp.dest('dist'));
});
```

❷ 流队列里的第一个元素是所有 Bower 组件的 CSS 文件

❸ 添加所有的库的代码

编译 LESS

❹ streamqueue 可以确保所有流的结果依然是有序的

正如其他 Node.js 的流相关的模块一样，streamqueue 也有一个 object 模式，正适合 Gulp 流使用❶。库❷的代码必须要优先处理。然后选择一个文件，这个文件里包含了通过 @import 引用的文件❸。cssimport task 会把所有引用的文件添加到主文件中，用文件的内容替换掉 @import 的声明。这样一来就能从头到文地把所有文件合并起来，得到一个最终文件。

正如合并流一样，流队列也是合并不同输入类型的好办法。区别在于，合并流可以大量并行处理，流队列是按照顺序来处理的。如果顺序对你的应用的代码非常重要，那就用流队列替换合并流吧。

### 7.3.3 通过 Gulp filter 来修改流的内容

流队列、穿插流、合并流都是用来把不同的输入添加到一个 Gulp 流中的技术。但是如果要把一个元素从流中移除该怎么做呢？让我们看一个例子。你有一个目录，其中有 ES2015 和 ES2016 的 JavaScript 文件，还有一些普通的 JavaScript 文件。你想用 Babel 编译前者，但是不用编译后者。由于 ES2015 和 ES2016 也是 JavaScript 文件，所以你想用 js 的后缀名替代 es，就像 7.2.1 节一样。这很有道理，因为 eslint 插件可以处理任何版本的 JavaScript，只要它是 JavaScript 就行（比如 CoffeeScript 就不行）。所以也许就没有必要使用两个插件了。然而，这就带来了一个问题，你并不想对那些不需要编译的文件执行 babel。尽管这样也不会崩溃，但是会拖慢速度。

你可以使用一个叫作 gulp-filter 的包来改变当前流的内容。你可以根据文件路径规则，筛选出一部分文件对象来传递给下一个 task。在之后的步骤中，你还可以恢复那些被过滤掉的元素，把它们重新添加到流中。假如你的 ES2015/ES2016 的代码是存放在不同的目录中的，或者有着不同的命名规范，你就可以用这个方法来筛选你要传

递的文件。下面的代码是具体的实现。

**代码清单 7-8 Gulpfile.js（节选）**

```
var filter = require('gulp-filter');

gulp.task('scripts', function() {
  const babelFilter = filter('*.babel.js', { restore: true });
  return gulp.src('scripts/**/*.js')
    .pipe(babelFilter)
    .pipe(eslint())
    .pipe(eslint.failAfterError())
    .pipe(babel())
    .pipe(babelFilter.restore)
    .pipe(uglify())
    .pipe(concat('main.js'))
    .pipe(gulp.dest('dist'));
});
```

❶ 在所有需要编译的文件的 js 后缀名前加上了“babel”

选择 srcipts 目录下所有的 ❷ JavaScript 文件

❸ 当编译完成后，恢复被过滤掉的文件

使用 Gulp filter 就可以过滤掉那些不适合管道中的下一步的文件❶。在同一个目录中，既有普通的 JavaScript，也有要编译的 JavaScript ❷。所有这些文件都可以用 ESLint 检查。这意味着你要把之前被过滤掉的文件恢复到流中❸。

你可以给一个流添加多个 filter。假设你的目录里还有第三方代码，而你不想把它们传给 ESLint。你的流应该像下面这样：

**代码清单 7-9 Gulpfile.js（节选）**

```
var filter = require('gulp-filter');

gulp.task('scripts', function() {
  const babelFilter = filter('*.babel.js', { restore: true });
  const vendorFilter = filter('!vendor/**/*.js', { restore: true });
  return gulp.src('scripts/**/*.js')
    .pipe(vendorFilter)
    .pipe(eslint())
    .pipe(eslint.failAfterError())
    .pipe(babelFilter)
    .pipe(babel())
    .pipe(babelFilter.restore)
    .pipe(vendorFilter.restore)
    .pipe(uglify())
    .pipe(concat('main.js'))
    .pipe(gulp.dest('dist'));
});
```

选中 scripts 目录下所有的 JavaScript 文件 ❶

过滤掉第三方文件 ❷

把所有过滤掉的文件添加回去 ❸

现在已经包括了第三方文件❶。你不想把它们传给 ESLint ❷。这就意味着把第三

方代码、普通 JavaScript 和 ES2015/ES2016 文件添加回流中❸。

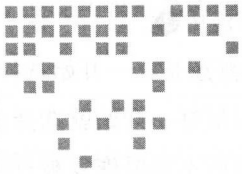
Gulp filter 非常适合用来在流中改变内容。但是最好还是在一开始选择文件的时候就准确一点，因为选择文件的时候需要读取操作，所以最好一开始就选择正确的文件，而不是先选择全部然后再删掉不需要的。如果你发现最后不需要恢复被过滤掉的元素，那说明你或许应该用一个更好的选择规则。

## 7.4 总结

在本章中，我们充分利用了流的强大功能，创建了非常复杂的流和操作：

- 穿插流可以在管道中插入新的输入。这让你可以在执行共同的 task 之前，先分别对不同的输入执行不同的编译操作。
- 合并流的作用也类似，但是功能更进一步。它不仅可以添加不同的输入类型，还可以在流合并之前，分别对它们执行不同的操作。
- 流数组和用参数创建的流就更加强大了。只需要定义一次流，就能根据不同的参数，产生多个输出结果，而不像之前那样只能产生一个输出结果，这个流可以在 task 中随意复用。
- 流的 snippet 可以把一些共同的操作打包成一个函数。这样这些操作就可以像一个 Gulp task 一样被添加到流的管道中了。在代码需要复用多次的时候，可以避免重复代码。
- 流队列可以保持输出结果的有序。如果你的代码需要一定的顺序，而且有些流耗时很长，那么它就非常有用。
- 在学会往流里添加文件后，你还学会了如何把不需要的文件从流中移除，只要使用 Gulp filter 就可以了。

既然你已经学会了流的各种使用方法，接下来让我们进入到 Gulp 最困难的部分：它庞大的插件生态系统。



Chapter 8 第 8 章

# 扩展 Gulp

**本章包括：**

- Gulp 插件生态系统。
- 如何处理那些对 Gulp 不友好的插件。
- 基于流的工具及其与 Gulp 的结合。
- Promise 及其与 Gulp 构建系统的集成。

JavaScript 构建工具通常会提供一些核心基本功能及其 API，其余工作则由各种各样的插件完成。这些插件本身其实也使用了其他工具的功能，只不过把它们包装了一层，使其能够和构建工具兼容。

Gulp 也不例外。正如你在前几章的例子中所看到的：CSS 预处理器 LESS 有一个 Gulp 插件叫 gulp-less，JavaScript 压缩工具 Uglify 则被包装成了 gulp-uglify。对于 Gulp 的创造者而言，这么庞大的插件系统非常有意义。相比于自己开发能满足各种边际条件的软件，不如给社区提供插件的功能，来快速、简单地集成新工具。

然而，这种强大的扩展性也有负面影响：新插件写起来越简单，冗余和重复的插件也就越多。截止本书写作的时候，Gulp 插件目录中已经有超过 1900 个插件，而且这个数字还在与日俱增。如何在众多看起来一模一样的插件中选出最合适的，这对开发者而言是个问题。比如说你想要选择压缩 CSS 的插件，你有下面这么多选择：gulp-

minify-css、gulp-cssmin、gulp-css 和 fd-gulp-cssmin。你究竟该选哪一个呢？选择的标准又是什么呢？

为了方便开发者选择，Gulp 的创造者给 Gulp 插件的好坏定下了明确的标准。简而言之，无论是对用户还是整个生态而言，都是使用的插件数量越少越好。如果插件太多，那么开发者就很难找到适合他的插件。很有可能有很多重复的插件，都是解决同一个问题的。有的插件的名字包含很多细节，可能它解决不了你的问题，但是却非常适合在某个特定场景下使用。

插件内容的不确定性和大量冗余重复的插件，这两个问题是 Gulp 的竞争者一直要面对的。而且，有的插件可能根本就不符合 Gulp 的架构（比如，完全无视 gulp.src 创建的流中的内容）。社区认为这些插件都是 Gulp 不友好的插件。

Gulp 的创造者甚至极力主张不要使用某些插件，而要用更直接的方法。他的意思就是使用 Node.js 生态系统中的那些既能解决你的问题，同时又能和 Gulp 无缝集成的包。这样就不需要任何插件了。

在本章中，我们树立了四个标准来帮助你遵循直接的方法：

- 如果一个插件不转换文件（无论是文件内容，还是文件路径），那这个插件就可以用别的东西替代。
- 如果一个插件做了不止一个任务，那么就应该用多个插件来替代。
- 如果一个工具是基于流的，那么你就不需要这个 Gulp 插件，直接使用这个工具就可以了。
- 如果一个工具会返回 Promise[⊖]，那么你就不需要这个 Gulp 插件，直接使用这个工具就可以了。

遵循着四条规则，可以让你的 Gulpfile 更加清晰，而且还可以让你更容易地找到正确的工具。准备好了吗？让我们到底什么是 Gulp 不友好的插件，以及如何识别它们。

## 8.1 Gulp 插件黑名单

当我们讨论 Gulp 插件时，我们常常会说某某插件是 Gulp 友好的。开发者可以给他们的插件贴上这个标签，然后 Gulp 就会在插件页面上展示这个插件[⊜]。这个标签不只

⊖ JavaScript 异步编程的一个概念，8.3 节会详细介绍。

⊜ http://gulpjs.com/plugins/。

可以用在 Gulp 插件上，还可以用在那些和 Gulp 一起使用且架构和 Gulp 类似的 Node.js 包上。这个标签不是 Gulp 团队的认证或者奖励，而是插件作者自己添加的。这也就是说，有些对生态系统有害的插件（因为它们没有遵循我们上面定义的规则），也可以添加上 Gulp 友好的标签。这就有问题了！一方面，你可以轻松地找出能和 Gulp 一起使用的包，即使它都不是一个插件；另一方面，由于可以每个包随意加上这个标签，所以多少也让这个标签变得没有意义了。

有鉴于此，Gulp 团队在 GitHub 上维护了一个 gulp 不友好的插件列表。这个黑名单的目的是过滤掉那些不应该被使用的坏插件，包括重复的、停止维护的或者在执行某些 Gulp 功能时崩溃了的插件。Gulp 团队对黑名单中的每个插件都会有简短的说明，解释它们为什么不应该被使用。这些说明就像是 Gulp 团队做出的判决理由，这些理由是多方面的，但是有三个理由却很常见。

### 8.1.1 理由：不是一个 Gulp 插件

黑名单中一个常见的判决理由就是“不是一个 Gulp 插件”。这个理由看起来很泛泛，但是它可以被总结成我们的第一个规则：“如果一个插件一个文件都不修改（无论是文件内容，还是文件路径），那么他就不是一个 Gulp 插件。”让我们以黑名单上的一个插件 gulp-compass 为例，它本质上就不是一个 Gulp 插件。

Compass 是 CSS 预处理语言 Sass 的一个扩展。Sass 和 LESS 很相似，但是它是基于 Ruby/C++ 的，而且语法上也有些区别。Compass 扩展包含了项目管理和其他的一些功能。有的开发者创造一个 Gulp 插件，来封装这些功能，但是却没有遵守核心的规则。下面的代码展示了 gulp-compass 的使用方法，让我们看一下。

**代码清单 8-1 Gulpfile.js 节选——Compass task**

```
gulp.task('compass', function()                    <—— 警告：不要在家里尝试
  return gulp.src('./src/*.scss')
    .pipe(compass({
      css: 'app/assets/css',                       | 这两行代码就是问题所在
      sass: 'app/assets/sass',                     | 这两行代码就是问题所在
      image: 'app/assets/images'
    }))
    .pipe(gulp.dest('app/assets/temp'));
});
```

上面标记的两行代码就是问题所在。根据文档，这两行的作用是告诉插件输入文件的目录（对象中的 sass 属性），以及输出文件的目录（对象中的 css 属性）。等一下！

为什么我们要告诉 Compass 输入文件和输出文件的目录呢？难道这不是 Gulp 自己就能做的事情吗？事实上，这个插件仅仅只是把自己放在了流里面，但是却不关心流的内容。这就是为什么 Gulp 核心团队认为它“不是一个 Gulp 插件”。

一个正确的、Gulp 友好的处理 Compass 的方式是使用 gulp-ruby-sass 插件直接来启用 Compass 扩展，如下代码所示。

**代码清单 8-2　Gulpfile.js 节选——Compass 不使用 Compass 插件**

```
var sass = require('gulp-ruby-sass');

gulp.task('compass', function() {
  return sass('./src/*.scss', { compass: true})
    .pipe(gulp.dest('app/assets/css'));
});
```

sass 返回了包含虚拟文件对象的流，就和 gulp.src 做的一样

compass 属性启用了 Compass 所需的一切。gulp-ruby-sass 和 Gulp 生态系统一致，返回了一组虚拟文件对象，可以被修改、转换和保存。注意这个插件不需要用 gulp.src 载入文件。这是因为 gulp-ruby-sass 只是一个对 Ruby 软件的简单封装，所以这个插件之前不能传入任何内容。所以，尽管它不可写，但是它能创建一个可读流，而且对 Gulp 非常友好。

## 8.1.2　理由：可以用别的东西替代

你在黑名单中经常可以看到类似“用 XY 替代”，或者“直接用 XY”之类的话。这依然和规则一有关。如果一个插件不修改文件或者流，那么就要重新评估这个插件的意义。可能有更直接的办法来完成这个插件的任务。

有的插件（像是 gulp-shell）很明显是用来执行 shell 命令的，如下面的代码所示。

**代码清单 8-3　Gulpfile 节选**

```
gulp.task('execute-shell', function() {
  return gulp.src('', {read: false})
    .pipe(shell('ls'));
});
```

这个插件是用来在 shell 中执行命令的，但是它不需要任何文件对象就能工作

事实上，这个插件的作者建议不要读取文件。这个插件不需要任何文件对象，而且完全没有用到 gulp.src 创建的可读流，这违背了规则一。

显而易见，这个插件没有遵循 Gulp 的规则。流在这里起了什么作用呢？有哪个文件需要转换？答案是没有。幸运的是，Gulp 插件黑名单还展示了一些同样能完成任务

的工具，它们并不需要流或者插件，如下面的代码所示。

**代码清单 8-4 另一个 Gulpfile.js 节选**

```
var exec = require('child_process').exec;   <-- 这是 Node.js 的核心模块

gulp.task('execute-shell', function (done) {
  exec('ls', function (err, stdout, stderr) {        <-- 使用正确的模块就可以直接执行 shell 命令了，不需要流或者文件
    console.log(stdout);
    console.log(stderr);
    done(err);                  <-- 告诉 Gulp 任务完成了
  });
});
```

所以要三思你是不是真的需要某个插件。有时候它只是对功能做了封装，但是这个功能可能可以直接在 Gulp 中使用，这样更加直截了当。其他例子包括我们在接下来的两节中介绍的工具。

### 8.1.3 理由：做的事情太多

还有一句话你经常会在黑名单的理由中看到："做的事情太多"。这种情况通常是：插件尝试解决的任务超过了一个。在 Gulp 中，要遵守单一职责原则（single responsibility principle）：一个 Gulp 插件只应该做一件事，并且把它做好。之所以提出这个原则，是有很多原因的。让我们以压缩 HTML、CSS 和 JavaScript 为例，来解释其中的原因。

#### 1. 不需要配置

Gulp 流不应该过度配置。开发者更喜欢使用那些不需要配置（如果可能的话）就能处理输入的工具。比如说，第 2 章里用的 minifyCSS 插件在压缩 CSS 时就不需要任何参数。

#### 2. 没有依赖

如果一个插件做的事情超过一个，比如说同时压缩 CSS 和 JavaScript，那么它很可能在 task 里封装其他 API，甚至封装 Uglify 和 MinifyCSS 等你已经用到的工具。如果其中一个工具修复了 bug 或者添加了新功能，插件的开发者就要更新插件所用的依赖。这会导致一大堆工作，而且往往会延误发布。有时候，插件开发者会不再更新依赖。如果把这个任务分成两个不同的插件，那么更新依赖的责任就转移到使用者头上了。

### 3. 子 task 松耦合

再次强调，如果你有一个同时压缩 CSS 和 JavaScript 的插件，万一你在移除一个子 task 时，不小心移除了这个插件，那么另一个使用这个插件的 task 也会崩溃。而且这种时候，很难看出来一个插件是否仅仅只在一个 task 里被用到了，谁会想到压缩 CSS 和压缩 JavaScript 用的会是同一个插件呢？

黑名单中有一个叫作 gulp-minifier 的插件。它就像我们之前说的一样，做了超过一件事情。它有很多其他工具的依赖，而且它不止接受一种类型的输入。如果让我们来处理，通常都会创建几个子 task 来实现。看看下面的代码。

**代码清单 8-5　Gulpfile.js 节选**

```
gulp.task('example', function() {
  return gulp.src('example/src/**/*').pipe(minify({
    minify: true,
    collapseWhitespace: true,
    conservativeCollapse: true,
    minifyJS: true,
    minifyCSS: true
  })).pipe(gulp.dest('example/dest'));
});
```

这个插件要设置很多配置才能最终工作 ❶

这个插件的过度配置违背了 Gulp 的原则❶。而且乍一看也看不出来到底是在处理 CSS，还是 JavaScript，还是两者都是。你可以看到这里的代码盘根错节，跟之前的示例代码比起来一点也不优雅。所以如果一个插件做了超过一件事，那就考虑用多个插件来分别完成它们。

## 8.2　集成其他基于流的工具

下一条规则：如果原本的工具是基于流的，那么就不需要额外的插件。Gulp 是原生支持流的。流式构建系统的核心理念就是通过流把虚拟文件对象传递给一个个操作。但是流还不仅仅如此，接下来你就会看到。

### 8.2.1　流、缓冲和 Vinyl 文件对象

在 Gulp 中，流是非常基本的概念，这一点我们在第 2 章已经充分解释过了。记住，流是持续流动的数据。它一块一块地传递，从某个地方输入，最后在某个地方输

出。比如你敲击键盘时，就创造了一个不断产生按键事件的流，流的内容最后会输出到某个文本中。

但是在 Gulp 中，你接触的最多的是对象的流，每一块数据都是一个虚拟文件对象（或者叫 vinyl 文件对象，vinyl 是虚拟文件系统的名字）。虚拟文件对象是硬盘里实际文件的替身，包含了文件的属性。参看图 8-1。

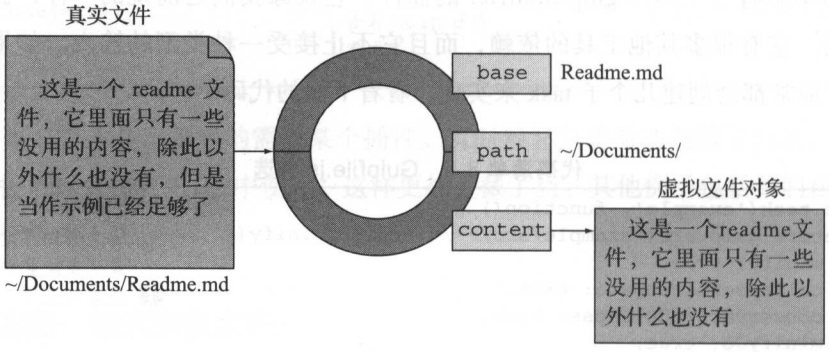



图 8-1 左边是真实的文件，它包含一些内容，有一个文件名，保存在一个目录中。硬盘上的文件可以转换成虚拟文件对象或者 vinyl 文件对象，vinyl 文件对象是对文件内容的轻量的封装，以便于快速地在内存中访问文件内容。另外，vinyl 文件对象还包含了文件的路径和名称

base 和 path 属性表示文件的名称，我们前几章里经常用到它。每次当你修改输出的文件名时（不管是合并，还是重命名，或者是转换），都必须修改这两个属性。base 属性只有文件的名称，而没有路径，便于快速地重命名文件；path 属性一般在你把一个文件从虚拟文件系统中保存到真正的文件系统中时（通过 gulp.dest），才会被修改。

另一个属性 content，则更加有趣。它可以有两种类型：缓冲（buffer）或者 Node.js 流。两者都可以转换成别的类型。

### 1. 流

一般来说，流在 Node.js 里面就是持续的数据流。这个数据可以一块一块地被读取（每一块的大小取决于这个流本身）。

Gulp.js 流是基于对象的，每一块代表流中的一个文件（就是之前提到的虚拟文件对象）。虚拟文件对象的内容也可以是某种流。但是，我们接下来要讨论的是可读文本流。它背后的概念是一样的：你可以一块一块地读取并转换数据，然后传递给下一个步骤。参看图 8-2。

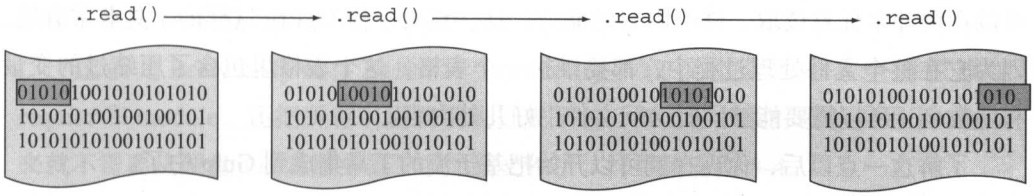



图 8-2　如果一个 vinyl 文件对象的内容是以流的形式保存的，那么你就可以很方便地一块一块地读取数据。你可以读取一小块，然后转换它，然后再处理下一小块。不过尽管它非常快，而且不是很占内存，但是它只能按照一个方向来读取数据

可读文本流可以非常快地处理数据，因为每次你处理的都是一小块（通常都是按行处理文件的），而且上一块处理好后就能把空间让给下一块了。但是这有一点局限性：每当一块数据已经处理好，并且你已经开始处理下一块时，你就不能再重新读取上一块数据了。设想一下，一个 Gulp 插件要把文件里所有内容转换成大写。每当你把一个字符转换成大写后，你就不再关心这个字符了，转而处理下一个了。

### 2. 缓冲

正是因为上面这个原因，所以有时候你需要缓冲。缓冲是在计算机中预留的一部分内存，可以储存一定量的数据。这些数据在内存里可以随意访问，有一个指针会给你想要的数据。图 8-3 展示了这一点。

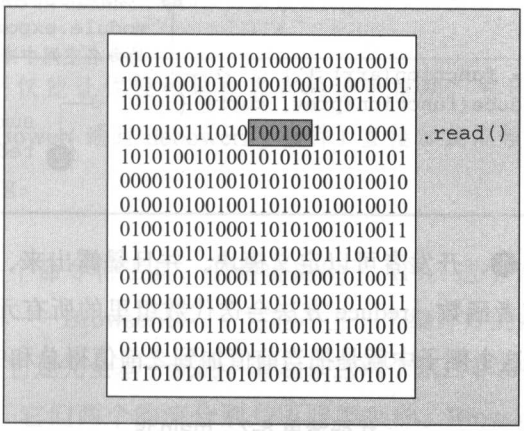



图 8-3　缓冲也是一种访问文件内容的方法。它需要加载整个文件，但是一旦加载完成，你就可以任意读取文件的内容，并且任意地修改。对于那些需要把处理后的结果和处理前的内容结合起来的插件，这一点是非常关键的

这个过程没有流那么快，因为你不能立刻地抛弃你不再需要的结果。但换来的是

可以在缓冲中任意读取、修改内容的能力。这一点对于像 Uglify 这样的工具非常有用，因为它在整个文件处理过程中，都要维护一个表格，这个表格里包含了压缩过的变量和函数名，所以需要能够同时访问文件中好几处内容。

了解这一点以后，你现在就可以开始把基于流的工具集成到 Gulp 中了。

### 8.2.2 集成 Browserify

了解流和缓冲后，现在你就可以给 Gulp 添加基于流的工具了。在这方面，有一个工具非常适合当作例子，那就是 Browserify。

Browserify 最近在开发者社区中造成了轰动。模块化和依赖解决方案一直以来都是 JavaScript 的一个问题，社区也产生了很多解决方案。Browserify 发明了一个完全基于工具本身的解决方法，不需要 JavaScript 开发者学习任何新东西。它仅仅是把 Node.js 处理模块的方法加以修改，变成了浏览器也能用的方法。

所以 Browserify 创造一种方法，能够在开发客户端应用时像在 Node.js 里那样引用模块。这种方法你已经在 Node.js 里用过了，应该很熟悉。下面的代码既可以在 Node.js 里用，也可以在 Browserify 里用。

**代码清单 8-6 sum.js**

```
module.exports = function(arr) {
  return arr.reduce(function(prev, curr) {
    return prev + curr;
  }, 0);
}
```

❶ CommonJS 环境里可以通过把 module.exports 添加到全局命名空间中来定义模块

❷ sum.js 通过数组的 reduce 方法来求和

通过 CommonJS ❶，开发者可以定义模块，并且暴露出来，供其他模块引用。你可以暴露值、对象或者函数。reduce 方法会迭代数组里的所有元素，并对它们执行一个定义好的方法。在这个例子中就是把当前值加到之前值得总和中。

**代码清单 8-7 main.js**

```
var sum = require('./sum');
console.log(sum[1, 2, 3, 4, 5]);
```

引用之前定义的模块，使用其中暴露出来的函数

浏览器不支持像 require 这样的方法，也不支持像 module 这样的全局对象。浏览

器也不能像 Node.js 那样从文件系统中加载模块。

这就是 Browserify 诞生的原因。Browserify 不仅仅往全局命名空间里添加了 require 和 module，还把所有引用的模块打成了包，浏览器加载好就能使用了。这样一来就不需要从硬盘（或者服务器）中加载文件了。详细请看图 8-4。

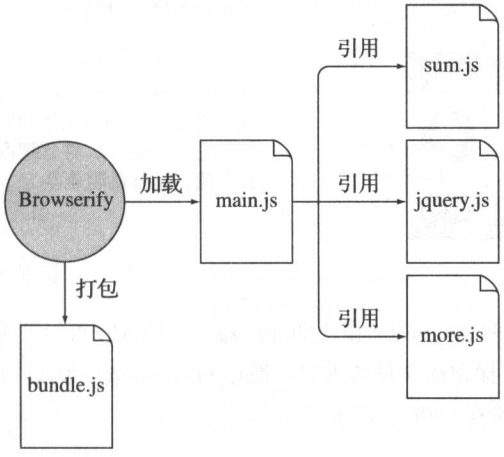



图 8-4 Browserify 是如何工作的。所有的 JavaScript 模块都可以用 require 或者 module.exports 来声明。Browserify 加以识别，知道要把哪些模块打包。它会打包所有代码，解析依赖，还能添加 require 和 module.exports 方法来让浏览器支持模块



注意 Browserify 不仅能让你轻松地在客户端使用模块，它还能让你从 NPM（本来就支持）和 Bower（通过 debowerify 插件）直接加载依赖。详情请看文档 http://browserify.org。

Browserify 有一个命令行工具，但是也提供了 API。这个 API 会产生一个流。现在我们有了一个工具——Browserify——它能产生一个编译好的内容的流；还有一个构建系统——Gulp——它是完全基于流的。看起来这两个东西非常适合组合起来。

唯一的问题是，它们两个的流分别是不同类型的。Browserify 产生的是文本流，Gulp 使用的是虚拟文件对象流，正如 8.2.1 节和第 2 章所提到的那样。但是如果你仔细看看虚拟文件对象的构成，就会发现虚拟文件对象里的 content 属性基本上也是一个文本流，这就是二者的相似之处。

Browserify 的流和虚拟文件对象的 content 流是类似的。现在还差文件名和路径名

称，就有一个完整的虚拟文件对象了。要创造一个虚拟文件对象，你可以使用一个叫作 vinyl-source-stream 的 Node.js 模块。参看图 8-5 的说明。

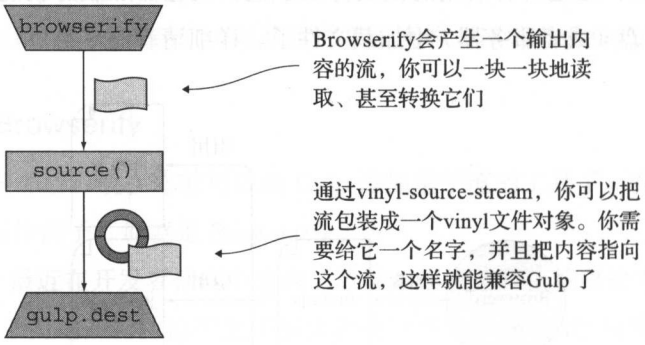



图 8-5 Browserify 是处理 JavaScript 模块的处理器。它的 API 可以产生一个流，供进一步使用，这个流包含了要保存的文件的内容。通过 vinyl-source-stream 包，你可以把它包装成一个 vinyl 文件对象在 Gulp 中使用

对应的代码非常直接明了，如下所示。

**代码清单 8-8 Gulpfile.js 节选**

```
var browserify = require('browserify');
var source     = require('vinyl-source-stream');

var bundle = browserify({
  entries: ['app/scripts/main.js']
});

gulp.task('scripts', function() {
  return bundle.bundle()
    .pipe(source('main.min.js'))
    .pipe(gulp.dest('dist/scripts'));
});
```

Browserify API 可以定义要打包的代码的信息

它接受要编译的入口文件作为参数，类似 gulp.src

bundle 方法会创建一个编译好的内容的流

由于它是一个流，所以完全可以直接传递 ❶

这里我们通过 vinyl-source-stream 模块来传递编译好的文件内容❶。这个模块会把文本流作为虚拟文件对象的内容，还会接受一个参数作为文件名。这样，你就可以把任何基于流的工具所产生的结果，保存在硬盘上的某个文件中了。由于 Gulp task 的机制需要你返回一个流，所以你可以用 Gulp 执行任何基于流的工具，即使这个工具最后不创建任何文件也没问题。

### 8.2.3　转换内容

用 Browserify 打包完 JavaScript 后，你应该抓住机会用打包好的 JavaScript 做些什么事情。尽管 Browserify 包办了所有打包工作，但是其中的内容还是有些冗长，因为所有的 JavaScript 代码其实没有怎么修改。你还是要用 Uglify 来压缩大小。但是内容的形式经过 source 转换后变成流了，而 Uglify 只能处理缓冲，正如前面 8.2.1 提到的那样。幸运的是，通过 vinyl-buffer 这个虚拟文件系统的模块，你可以把流中的内容转换成缓冲，如图 8-6 所示。

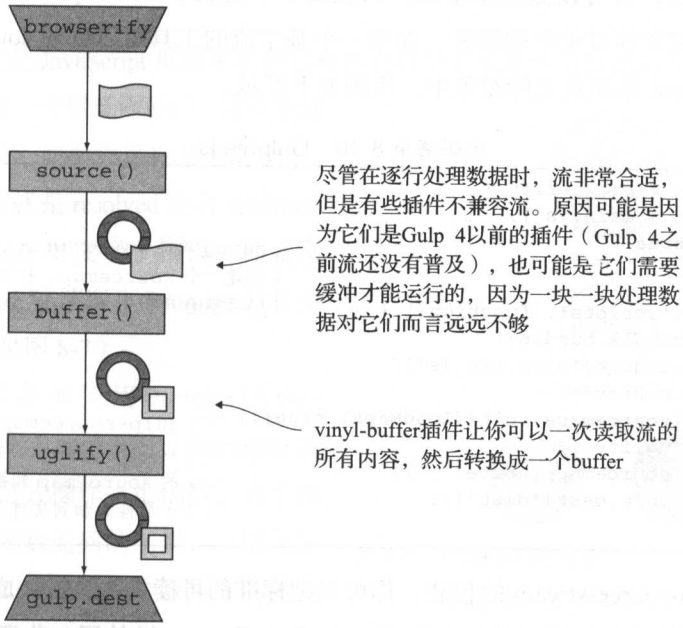



图 8-6　gulp-uglify 是用来压缩 JavaScript 的插件。它要转换整个文件的代码，就必须要能够操作文件的所有内容。因此，gulp-uglify 需要缓冲才能执行。通过 vinyl-buffer 插件，你可以把 vinyl 文件对象的内容从流转换成缓冲，来让它兼容这类插件

相关的代码如下。

**代码清单 8-9　Gulpfile.js**

```
var browserify = require('browserify');
var source     = require('vinyl-source-stream');
var buffer     = require('vinyl-buffer');

var bundle = browserify({
  entries: ['app/scripts/main.js']
});

gulp.task('scripts', function() {
```

```
  return bundle.bundle()
    .pipe(source('main.min.js'))
    .pipe(buffer())
    .pipe(uglify())
    .pipe(gulp.dest('dist/scripts'));
});
```

vinyl-buffer 插件把 content 属性从流转换成缓冲，然后给 Uglify 使用

vinyl-source-stream 和 vinyl-buffer 都是用转换或者创建虚拟文件对象的工具。vinyl-source-stream 创建一个新的文件对象。vinyl-buffer 修改 content 属性。熟悉了文件对象的构成后，就可以做更多事情。回想第 6 章集成 sourcemap。sourcemap 只是另一个添加到虚拟文件对象中的属性。如果一个基于流的工具可以创建 sourcemap，你就可以把 sourcemap 添加到文件对象中，代码如下所示。

**代码清单 8-10 Gulpfile.js**

```
var bundle = browserify({
  entries: ['main.js'],
  debug: true
});

gulp.task('scripts', function() {
  return bundle.bundle()
    .pipe(source('main.min.js'))
    .pipe(buffer())
    .pipe(sourcemaps.init({loadMaps: true}))
    .pipe(uglify())
    .pipe(sourcemaps.write('.'))
    .pipe(gulp.dest('dist'));
});
```

debug 属性告诉 Browserify 要创建一个 sourcemap，打包完代码后就会产生一个 sourcemap

gulp-sourcemap 插件和 loadMaps 选项会把产生的 sourcemap 转换成一个属性添加到文件系统里

通过 vinyl-source-stream 的包装，你可以把标准的可读文本流转换成基于虚拟文件对象的流。这样一来，就能让它们兼容 Gulp 和一些 Gulp 插件了。你所创建的流是全新的，它的内容来自原本的工具产生的文本流（在上面的例子中是 Browserify）。但是 Gulp 插件不仅能处理流的形式的内容，还可以处理缓冲的形式的内容。Vinyl 文件系统提供的 vinyl-buffer 就非常适合处理这种插件。

Browserify 是展示如何把基于流的工具集成到 Gulp 中的非常好的例子。Browserify 基于流工作，和 Gulp 一样。所以这两个工具在处理内容时有很多共同点。如果一个工具是基于流的，那就赶紧用轻便的 vinyl 模块封装它们，加入到 Gulp 生态中吧。

## 8.3 集成基于 Promise 的工具

我们要解释一下最后一条规则“如果一个工具会返回 Promise，那么你就不需要这

个工具的插件”，以免你还没听说过 Promise。总体而言，Promise 是一种用同步的方法来处理异步结果的方法。听起来很难懂？好吧，也许是有点。先剧透一下：Promise 可以非常好地集成到 Gulp task 系统中。这一章就会告诉你什么是 Promise，以及它的用法。

### 8.3.1　Promise 的用法

在 JavaScript 中，异步操作和方法非常普遍。比如说处理事件、从服务器异步加载资源等等。这些计算有一个共同点：它们的结果不会立刻返回，而是会在将来某个时刻返回。

Promise 就是 JavaScript 里用来处理这种你暂时没有获取到的数据的方法。Promise 顾名思义，就像是一个保证你最后一定会拿到数据的“承诺”。如果你最后拿到数据了，Promise 的状态就是 resolved 或者 fulfilled。但是如果 Promise 里发生了错误，你就拿不到数据了，这种情况下 Promise 的状态就是 rejected（见图 8-7）。

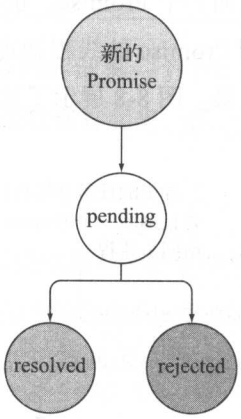



图 8-7　Promise 的状态。当一个 Promise 刚刚创建时，它的状态是 pending，意思是它异步计算的结果还没有返回。当它返回后，Promise 的状态就变成 resolved 了。如果发生了错误，那就是 rejected

让我们看看如何用 Promise 的 then 把异步操作变得“thenable”。thenable 的意思是你可以链式操作一堆函数，每个函数处理上一个函数返回的结果，一般是异步的结果。比如说要写一个数组求和的函数，那么原生数组方法 reduce 就足够了，它能把数组里的所有值 reduce 成一个。但是这样的代码是同步阻塞的。把它包裹在 setTimeout 里就能让它变成异步的，但是副作用就是需要回调才能处理结果。这种场景就非常适合 Promise，代码如下所示。

**代码清单 8-11　Promise 例子**

```
function sum(arr) {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      try {
        var sum = arr.reduce(function(prev, curr) {
          return prev + curr;
        }, 0);
```

❶ 创建一个新的 Promise

❷ 把代码包裹在 setTimeout 里

```
        resolve(sum);
      } catch(err) {
        reject(err);
      }
    }, 0);
  });
}
```

一旦 sum 计算好了，就 resolve 它；Promise 的状态也就变成了 resolved

如果发生了错误，可以捕获这个错误，然后状态就会变成 rejected

主要想法是把 sum 函数异步地执行❶。Promise 接受一个回调函数作为参数。这个函数有另外两个函数作为参数：resolve 和 reject。当你创建了一个 Promise，Promise 的状态就是 pending。把它包裹在 setTimeout 中能异步执行函数❷，即使我们把 timeout 设为 0 也一样。这样一来就能保证一开始调用 sum 的主线程可以继续执行下去。

sum 返回一个 Promise，也就是说你可以在结果暂时没有返回的情况下，继续执行代码。调用 Promise 的线程知道未来某个时刻你会拿到结果，并且知道你会在 then 中处理这个结果。图 8-8 展示了这行过程的时间轴。

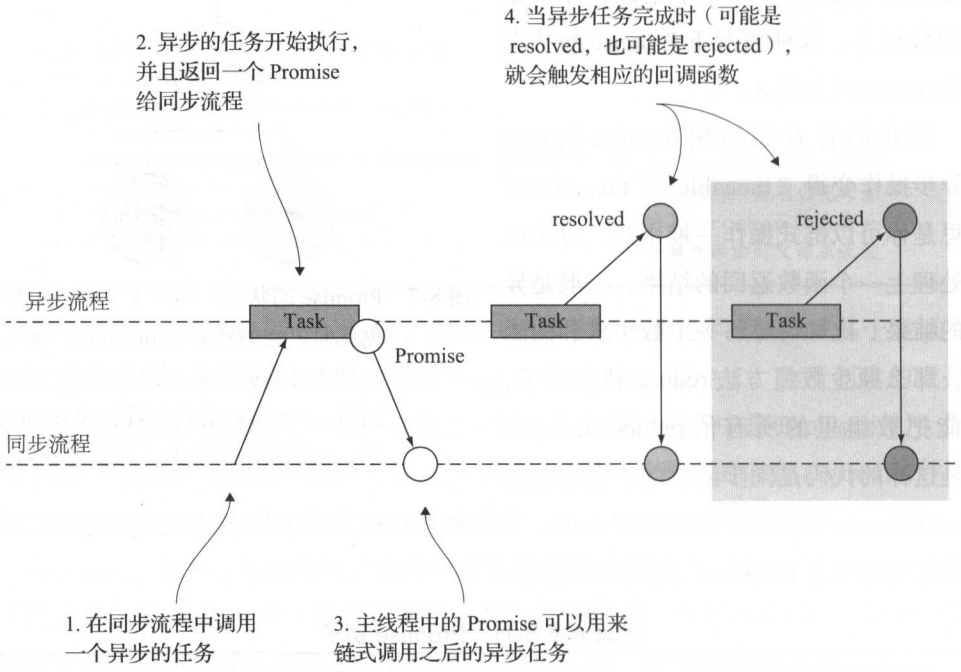



图 8-8 Promise 执行的时间轴。同步流程调用一个函数，这个函数返回一个 Promise（1 ~ 2）。当这个函数执行的同时（2），你在同步流程就已经得到了 Promise，可以处理结果了（3）。在某个时刻（4），Promise 会 resolved 或者 rejected，然后就会触发相应的下个步骤，或者出错

还是接着上面那个例子。sum 函数给你了一个 Promise，通过这个 Promise，你最后会得到结果；然后你就要处理这个结果。具体怎么处理，请看下面的代码。

**代码清单 8-12　调用 sum 函数**

```
你想求这个数组的和；这个
方法返回一个 Promise
  sum([1,2,3,4])
    .then(function(result) {
      console.log(result);
    })
    .catch(function(err) {
      console.error('There has been a problem');
    });
```

假设这个 Promise resolved 了，你想对结果做些事情，比如说打印出来

假设这个 Pormise rejected 了，你想捕获这个错误，并进行处理

Promise 允许你链式执行一系列步骤，并且把异步返回的结果一个个传递下去。你应该对这个链式调用很熟悉了，它和你之前在流里面传递数据看起来一模一样。

## 8.3.2　在 Gulp task 系统中使用 Promise

Promise 的特性——链式异步执行任务，并且处理返回结果——让它完美适合 Gulp task 系统。gulp.task 完全支持 Promise。更棒的是，你已经在 Gulpfile 里用过它了。在第 3 章里，你用到了 del 模块。del 不仅删除了文件，最后还返回了一个 Promise，如下面的代码所示。

**代码清单 8-13　Gulpfile.js**

```
gulp.task('clean', function() {
  return del('dist');
});
```

Gulp task 系统不仅能处理流，还能处理 Promise。结构都是差不多的：流要么会结束，要么会报错，类似于 Promise 的 resolve 和 reject。

在哪种 task 里 Promise 要优于流呢？Gulp 流在处理和文件相关的东西时非常好，所以在不处理和文件相关的东西时，Promise 或者基于 Promise 的工具就是一个不错的替代。举个例子：你想检查某个保存在 package.json 里 Gulp 插件是不是在黑名单中。

Gulp 把所有黑名单保存在 GitHub 的一个文件中。通过域名你可以获取到这个文件的内容。通过使用标准的 fetch 方法，你就可以得到一个 Promise 来处理返回的内

容。然后，你就可以可以拿这个文件和 package.json 作比较，看看插件是否在黑名单中。图 8-9 展示了这一过程。

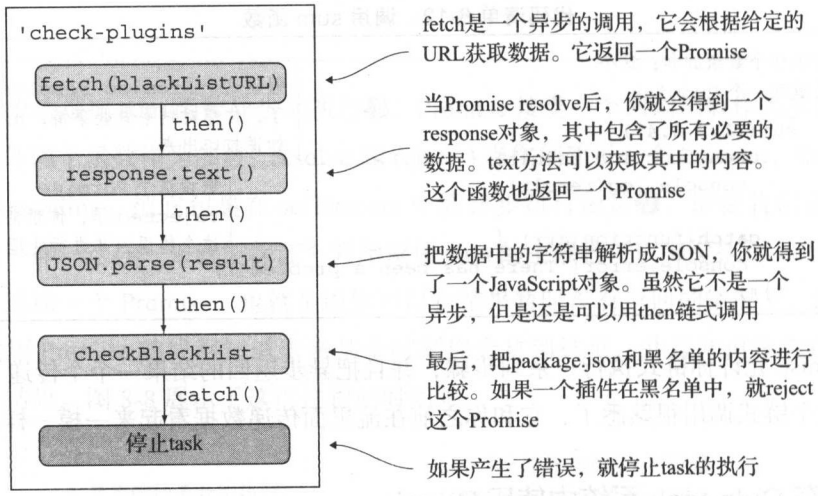



图 8-9 Promise 和 then 方法的使用。这个 task 和基于流的 task 非常类似

下面的代码是具体的实现，我们使用 node-fetch 作为 fetch 方法的事件。Promise 标准是 Node.js 4.0 的。

**代码清单 8-14 Gulpfile.js 节选**

```
var fetch = require('node-fetch');
var modules = require('./package.json');

var blackListURL =
  'https://raw.githubusercontent.com/gulpjs/plugins/master/src/' +
    'blackList.json';

gulp.task('check-plugins', function() {
  return fetch(blackListURL)
    .then(function(response) {
      return response.text();
    })
    .then(function(result) {
      return JSON.parse(result);
    })
```

node-fetch 是 Node.js 对 fetch 的一个实现

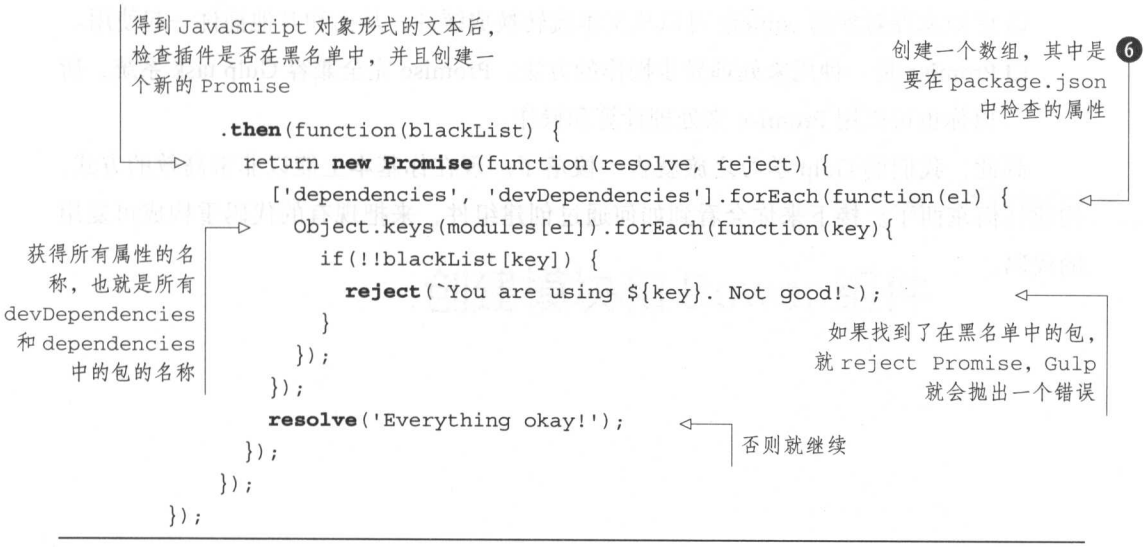
❶ 通过引入 package.json 加载 node 模块的信息

❷ 这个 URL 指向黑名单

❸ 返回的是一个 Promise，而不是流

❹ 得到 response 之后，返回文本内容

❺ 把文本内容解析成 JSON




由于这是一个 JSON，所以当你加载完后，会得到一个 JavaScript 对象❶。这个文件❷包括了所有黑名单中的插件。这里的 Promise 就是你从 URL 中将要获取的数据❸。这个函数❹也会返回 Promise。这个调用❺是同步的，但是当你用 then 链式调用下一个函数时，你还是可以在函数里面使用上一个函数返回的结果。最后检查一下 dependencies 和 devDependencies 是否包含黑名单中的插件❻。

这段代码有点长。不过，我们没有用任何 Gulp 插件，仅仅用基于 Promise 的工具创建了一个新的 task。但如果你想处理并转换文件，或许还是流更合适。

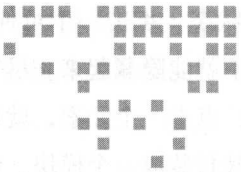
## 8.4 总结

在本章中，你了解了 Gulp 黑名单，还学会了不使用插件来创建 Gulp task 的几种方法：

- 有些插件是不符合 Gulp 的核心原则的。这些插件都在 Gulp 插件黑名单里。Gulp 的核心团队会定期维护和更新这个黑名单。
- 如果一个 Gulp 插件不修改流或者流的内容，那么通常这些插件都是不必要的，可以使用更直接的方法来处理 task。具体的替代品在 Gulp 插件黑名单中有。
- 你可以轻松地把其他基于流的工具，比如 Browserify 集成到 Gulp 的构建管道中。只需要添加几行代码，就能让文本流符合虚拟文件系统。

- 虚拟文件对象的 content 可以从文本流转换成缓冲，从而和其他插件一起使用。
- Promise 是一种用来处理异步操作的方法。Promise 完全兼容 Gulp task 系统，所以你也可以用 Promise 来处理计算和操作。

到此，我们的 Gulp 学习之旅就告一段落了。现在你基本上能以非常高效的方式，构建任何东西了。接下来你会看到如何通过创建组件，来把现有的代码重构成可复用的代码。



第 9 章 Chapter 9

# 创建模块和 Bower 组件

**本章包括：**

- 异步模块规范。
- CommonJS 模块系统。
- 通用模块规范。
- 暴露模组功能。
- 部署私有和公共组件。

开发的时候，我们经常会把某些功能封装成可以复用的模块。有的模块是函数，传入不同的输入，就能给出对应的结果。有的模块是对象或者类，包含了模块状态和行为。开发者想要的是能够复用提供的功能，并且通过测试来保证模块的质量。

模块封装了功能，并且对外暴露一个 API。其他模块不需要知道它的内部工作细节，就可以调用这个模块的 API，从而使用它的功能。如果说模块是在代码层面上封装功能，那么组件则是在代码的组织层面上的封装功能。组件是一堆模块的自给自足的集合。组件能够在代码之间独立地对外提供功能。模块也可以提供一定程度上的复用性，但是这种复用是在项目和项目之间复用。开发者应该能够以依赖的形式把组件添加到项目中，并且不需要怎么配置就能使用其功能。

比如说，做网页时有两个很常见的需求。点击一个菜单按钮，就能切换菜单的显

示和隐藏的状态。点击一个可伸缩面板（accordion）的标题，就能显示下面隐藏的内容，再点一下就能隐藏起来。尽管它们的控件和元素完全不同，但是它们内在的行为逻辑是一致的：点击一个元素，就可以切换其他元素显示和隐藏的状态。看起来似乎这个东西也应该被封装成一个模块，然后以组件的形式在未来的项目中复用。见图 9-1。

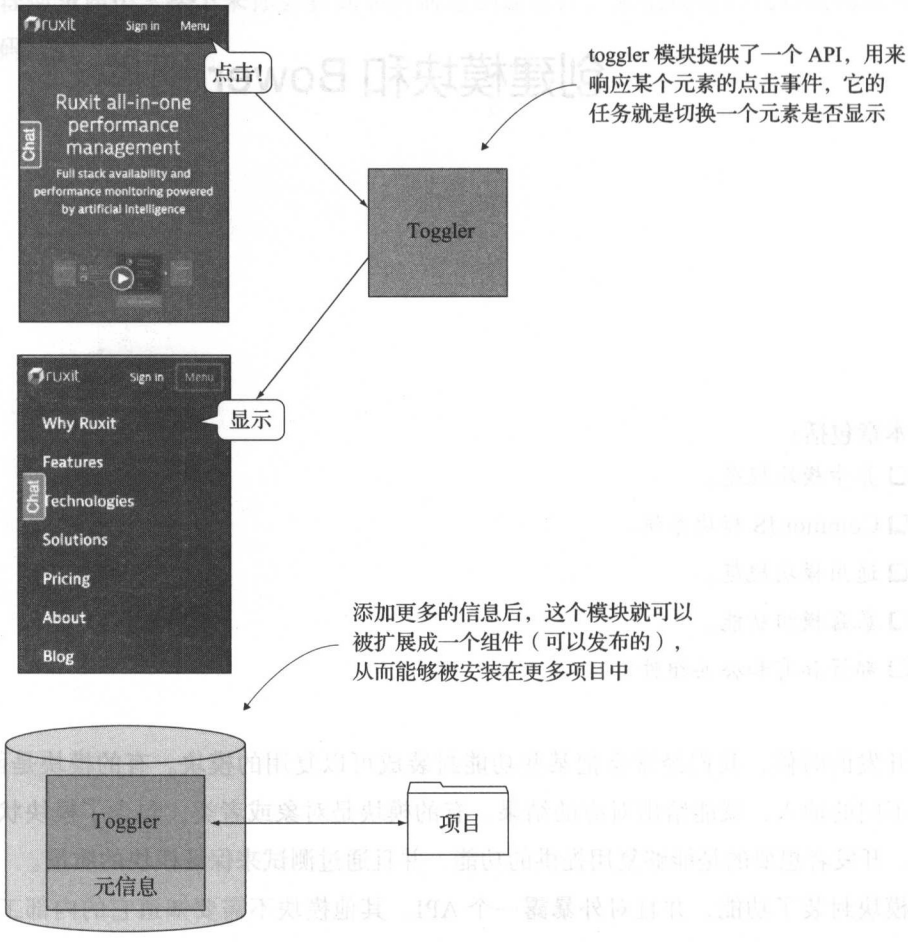



图 9-1 toggler 模块封装了一个功能，就是响应点击事件，然后切换另一个元素是否显示。它的 API 要接受不同的参数（元素要切换的状态），但是却能够完全独立地在应用中运行。添加更多信息后，我们就可以把这个模块扩展成一个可以发布的组件了

在本章中，我们会把这个 toggle 的 UI 模板重构成一个模块，让它可以作为一个 Bower 组件来复用。这样不仅可以让项目的架构更合理，还能把零碎和通用的代码从项目的核心代码中分离出来。这样软件中的独立的功能就可以整合到易懂的单元中了。

你将学会如何在 JavaScrip 中定义模块，还会知道要把这些模块当成依赖使用所需的必要的信息。然后你就可以像第 4 章中所介绍的那样把依赖当成可分发、可安装的软件添加到其他项目中去了。

**关于模块规范**

下面两节将会介绍 JavaScript 中最常见的模块规范。要使用模板不一定非要创建 Bower 组件，但是模块可以帮你更好地组织代码，从而更好地把代码集成到现有的应用中。由于本章的关注点在于尽可能简单地复用 JavaScript 代码，所以使用模块规范不仅仅只是推荐而已，而是必需的。

## 9.1　异步模块规范

和其他编程语言不同，JavaScript 长期以来都缺少定义模块的语法和加载模块的机制。这是因为 JavaScript 是运行在浏览器中的，而且加载机制是通过简单的 script 标签来指向源码。随着 Node.js 的诞生，JavaScript 可以运行在服务端了，同时客户端应用也越来越流行，因此 JavaScript 界随之产生了对优秀和健壮的模块系统的需求。

目前产生了很多在 JavaScript 中定义模块的规范，但是和其他前端的新事物一样，这些规范在各个平台的实现情况也不尽相同。在本书写作之时，还没有平台完全实现了模块规范。但这并不意味着你不能使用模块。JavaScript 标准一般都是来源于流行的库和已经实现的想法。

最常见的两个定义模块的方法就是 AMD 和 CommonJS。我们会接下来会讨论 AMD 和 CommonJS，以及它们的结合 UMD。

让我们从 AMD 开始，因为它不仅是最老的方式之一，而且它是为我们的运行平台浏览器所特别设计的。

### 9.1.1　AMD——为浏览器设计的模块定义规范

异步模块规范（Asynchronous Module Definition，AMD）是 JavaScript 界里最常见的两个定义模块的方法之一。AMD 是由 AMD JS 小组[⊖]定义的。这个小组的成员来自各个企业，为公共和开源领域开发。顾名思义，AMD 标准可以用异步加载技术来定义

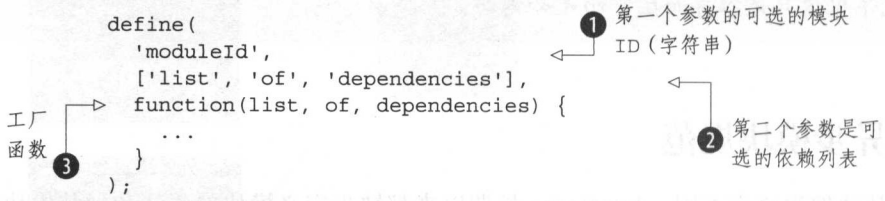
⊖ https://github.com/amdjs。

模块和依赖。这非常适合浏览器的环境，因为在浏览器里异步加载是包括在 HTTP 请求和 AJAX 中的。它还有其他功能，比如按条件加载那些可能只会在特定情况下用到的模块，而其他代码同时也在运行。这样一来，浏览器就只会在需要的时候，传输应用需要的文件和代码，而不会加载很多用不到的 JavaScript。

AMD 提供了两个方法用来定义和加载模块，让我们来看一下。

### 1. define 方法

define 方法，顾名思义是用来定义模块的。你可以把依赖和模块一起传递给它。唯一一个必需的参数是一个工厂函数，其中是模块的代码。define 的用法如下所示：




如果定义了 ID，那么这个模块就会以这个 ID 为名❶。否则，模块加载器就会以文件路径来识别它。第二个参数是依赖。它是一个数组，元素是字符串，每个字符串都指向一个文件（不包含 .js 后缀）❷。工厂函数❸里包含了模块的代码。工厂函数最后返回的结果，就是要暴露给其他模块的功能。工厂函数的参数是依赖的模块所暴露的 API，顺序和第二个参数的顺序相同。

所以说，用 AMD 定义的模块至少需要一个工厂函数来暴露 API 给其他模块使用。依赖数组中的依赖会被异步加载，如图 9-2 所示。

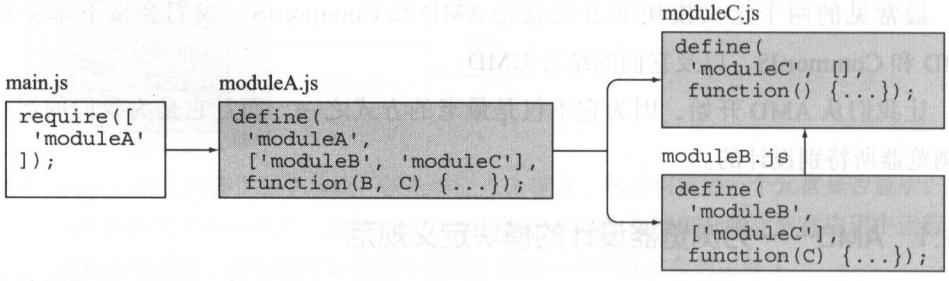



图 9-2 AMD 模块加载流程示意图。每个箭头都表示一个加载的落成。在 main.js 文件中，我们加载了一个模块 moduleA。它有两个依赖的模块：moduleB 和 moduleC。moduleB 同时也依赖 moduleC。AMD 是通过函数和回调来从网络异步加载的，回调函数会把 API 暴露到对应的参数上

模块的加载是通过客户端（浏览器）向服务器发送 HTTP 请求来完成的。这个传输过程要消耗大量时间。如果一个模块已经传输过了，那么就不会再次传输了，因为它的内容已经可以用了。

AMD 既可以定义具名模块，也可以定义匿名模块。具名模块可以通过开发者定义的名字来加载，匿名模块则会隐式地以文件名来加载。

### 2. require 方法

require 方法和 define 方法相对应。define 方法是用来定义模块和依赖的，require 方法用来显式地加载模块。它可以用来作为一个应用的入口（entry point）。图 9-3 展示了使用 require 时如何根据条件来加载模块。

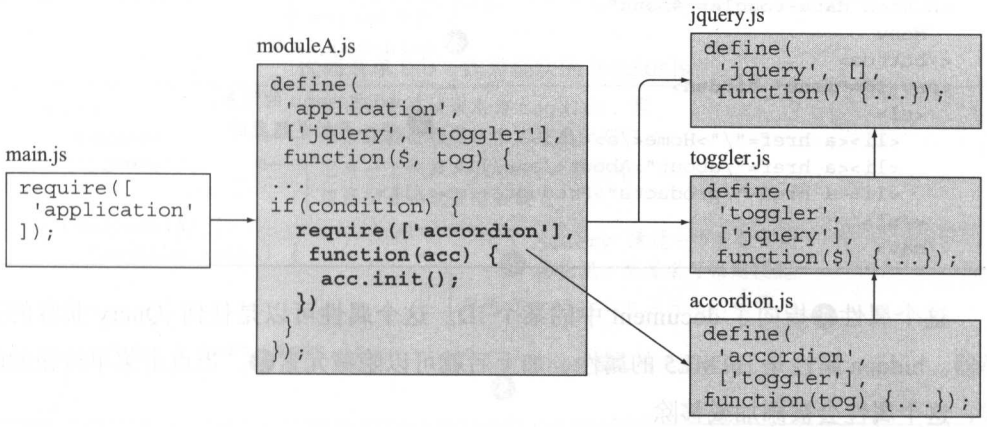



图 9-3　按条件加载更多模块。你可在代码的任何地方使用 require 来加载另一个模块。require 调用会发送一个请求来下载模块。由于这个请求是异步的，所以需要一个回调才能拿到暴露的 API

由于加载是异步的，所以只有在回调函数里才能获取新拿到的 API。

```
require(['jquery', 'toggler'],      <—— require 的模块
  function($, tog) {                <—— 回调函数，两个模块都可以在里面使用
    ....
  }
);
```

define 和 require 在 JavaScript 运行时里是不存在的，它们由 AMD 的具体实现——模块加载器（module loader）来提供。最流行的一个实现是 require.js，它需要一些设置，但是它的文档（http://requirejs.org）非常棒，可以帮助你开发完整个项目。

### 9.1.2 重构到 AMD

接下来让我们看看如何把一个用标准的 jQuery 代码编写的例子，重构成一个 AMD 模块。我们想要创建可以复用的模块，它的功能是改变 HTML 元素的状态。比如网站利用菜单按钮来切换菜单的显示状态，或者是点一下标题就能展开下面的隐藏内容，再点一下就重新隐藏之类的。虽然菜单和可扩展面板的 UI 构成不一样，但是它们的行为是一样的：点击显示，再点隐藏。当你写好代码后，你的代码可能就与代码清单 9-1 和 9-2 差不多。

你要创建的组件依赖 jQuery，然后会查找所有带有 data-toggle 属性的标签。data-toggle 属性指向 document 中的另一个元素，这个元素就是要切换显示状态的元素。

**代码清单 9-1 HTML 示例**

```
<button data-toggle="#menu">
  Menu
</button>
<nav id="menu" hidden>
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/about">About</a></li>
    <li><a href="/products">Products</a></li>
  </ul>
</nav>
```

❶ 这个 button 有 data-toggle 属性

❷ id 为 menu 的元素，它一开始是隐藏的

这个属性❶指向了 document 中的某个 ID。这个属性可以是任何 jQuery 兼容的选择器。hidden 属性是 HTML5 的属性，加上后就可以隐藏元素❷。当点击菜单按钮的时候，这个属性会被添加或移除。

**代码清单 9-2 togger.js**

```
function init() {
  $('[data-toggle]').each(function(idx, el) {
    var $el = $(el);
    var $target = $($el.data('toggle'));
    $el.on('click', function(e) {
      e.preventDefault();
      if($target.attr('hidden')) {
        $target.removeAttr('hidden');
      } else {
        $target.attr('hidden', true);
      }
    });
  });
}
```

调用这个函数来初始化

每个元素都必须要用 jQuery 封装一下

❶ 用 jQuery 选择器选中每个带有 data-toggle 属性的元素

❷ 根据 data-toggle 属性里的选择器，选择目标元素（带有 hidden 属性的元素）

在按钮上绑定的 click 事件回调会检查 hidden 属性是否存在。如果不存在，就添加上去，反之亦然

第 2 行❶会返回一组 jQuery 元素，你要用 each 方法来迭代处理。你可以使用 jQuery 的 data 函数来获取到选择器❷，并且由于它返回的是一个字符串，你要再用 jQuery 选择器来根据他找到目标元素。

这些代码对于创建模块已经够了，因为它已经满足了我们之前提到的几个特点：

- API 暴露了一个 init 函数，供开发者调用。
- 有一个依赖，jQuery。
- 它的功能完整且可以复用。

我们要把这些代码重构成一个符合 AMD 规则的模块。如 9.1.1 节所示，AMD 引入了两个概念：一个用来定义模块的 define 方法，以及一个用来加载模块的 require 方法。首先，我们需要第一个方法 define 来封装代码的功能。详情请看下面的代码。

**代码清单 9-3　作为模块的 toggler.js**

```
define(
  'toggler',
  ['jQuery'],
  function($) {
    function init() {
      ...
    }

    return { init: init };
  }
);
```

显式地把模块命名为 toggler，模块加载器会根据这个名字来识别它

它有一个依赖，jQuery；模块处理器知道要从哪里加载它

jQuery 通过一个变量传到函数里，变量名可以随便起 ❶

原来 JavaScript 文件里的代码

把 API 暴露给其他模块 ❷

由于我们一直都是用 $ 来表示 jQuery，所以这里还是用它❶。当开发者 require toggler 模块时，只要调用 init 方法就可以了❷。

在 AMD 的模块的帮助下，我们封装一个功能，并且使其在整个项目里都可以复用。定义和加载模块都是异步的，依赖像 RequireJS 这样的工具来执行。下面的代码展示的是 RequireJS 基本的设置。

**代码清单 9-4　main.js**

```
requirejs(['toggler'])
```

这个 main.js 文件除了一行加载前面定义 toggler 模块的指令外什么都没有了。其他部分由 RequireJS 来完成：加载 JS 文件和 jQuery 依赖。为了避免命名的冲突，require.js 使用了 requirejs 替代 require 作为加载函数的名字。这也可以在配置里修改。

在 HTML 里加上 require.js 库和要加载的文件的链接。看起来如下：

```
<script data-main="scripts/main.js" src="scripts/require.js"></script>
```

这样就会异步模块加载就会启动了。Require.js 加载 main.js，再加载 toggler.js，再加载 jQuery，就像图 9-3 里的一样。然而，异步有时候会让人搞不明白。下一节里就会介绍一种同步的模块定义规范。

## 9.2 CommonJS 和通用模块规范

尽管 AMD 是专门为浏览器设计了，但是它的加载机制有点复杂，让部分用户感到困惑。这就催生出了一个更简单的格式：CommonJS。它的不同之处在于，你不再需要思考异步加载模式了，引入一个模块就和调用一个函数一样简单。它的基本原理是事先编译代码，创建一个比原来更大的代码包，其中包括了所有应用运行所需的代码。它相比于 AMD 的缺点在于，你不能按需加载特定的功能了。

随着这两个模块定义规范的流行，开发者必须要让组件同时能够支持两个规范。这又催生出了一个新的规范叫作通用模块规范（Universal Module Definition，UMD）。让我们来看一看这两个规范。

### 9.2.1 CommonJS 模块

CommonJS 模块是第二流行的模块定义规范，最近的人气更是持续上涨，因为它的工具效率更高、语法更简单。一开始，CommonJS 模块是 JavaScript 服务器模块的规范。注意了，它就是你之前在 Gulpfile 和 Browserify 里用过的 module.exports/require 组合。你应该对这个用法很熟悉了。

```
var $ = require('jquery');
function init() {
  ...
}
module.exports.init = init;
```

把依赖加载到当前文件中

module export 对象获得了一个新属性，它指向 init 函数

这个模块现在就可以用了，和之前的例子差不多，区别在于没有回调函数，它是同步的过程。同时也没有必须要定义依赖，因为依赖在被引用的时候，会一同被加载好。由于省略了回调和依赖，代码变得更简洁、更易懂了，尤其是对于那些不熟悉 JavaScript 异步的人而言。因此，CommonJS 逐渐变成了流行的模块定义规范。

但是对于加载我还是要提一句。CommonJS 的加载机制是完全同步的，也就是说每个 require 语句都会短暂阻塞代码的运行，直到模块加载完毕。图 9-4 展示了这个过程。

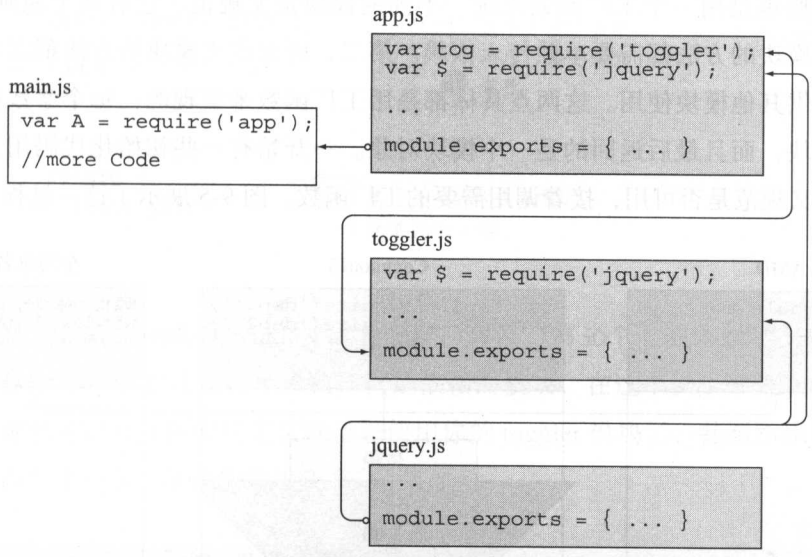



图 9-4　和 AMD 异步的特性不同，CommonJS 模块直接从模块提供暴露的 API，让开发者用同步的方法使用它们。每个箭头都可以当成是一个加载过程，但是不是通过网络加载，而是从内存或者文件系统加载。这个过程很快，而且一般不会阻塞

对于像 Node.js 这样的系统来说，这完全没有问题，因为它读取文件很快。对于 Browserify 也没有问题，因为所有必需的模块都已经打包成一个 JavaScript 文件了。然而，这也就意味着，原生的 CommonJS 模块不适合浏览器，因为浏览器的加载机制不支持同步（毕竟 AJAX 的第一个 A 代表 Asynchronous(异步)）。让 CommonJS 在浏览器中运行的唯一办法就是用像 Browserify 这样的工具把所有模块都打包成一个 JavaScript 文件。

## 9.2.2　通用模块规范

AMD 和 CommonJS 在社区里都很常用，各种各样的项目都会用到它们。这二者选择的标准通常取决于项目的要求和应用的架构。这个应用能不能分割成好几个按需加载的小模块呢？你是否愿意到时候部署整个 JavaScript 应用，而只把模块当成是组织代码的工具呢？在有的情况下，你可能会放弃使用模块定义规范，直接使用 JavaScript

的全局命名空间。这些决定都是因项目而异的，所以上面这三个选项都不能排除。由于这两个模块定义规范都有各自的使用场景，社区想出了一种整合这两个模块定义规范的方法，叫作通用模块规范（Universal Module Definition）。

基本原理是用一个工厂函数来统一不同的模块定义规范。它有两个原则：第一，所有定义模块的方法都需要单独传入依赖。第二，所有定义模块的方法都需要返回一个对象，供其他模块使用。这两点具体都是用工厂函数来实现的：每个参数都是一个依赖的模块，而且最后返回的是一个模块对象。一开始有一些初始化代码用来检查各个模块定义规范是否可用，接着调用需要的工厂函数，图 9-5 展示了这一过程。

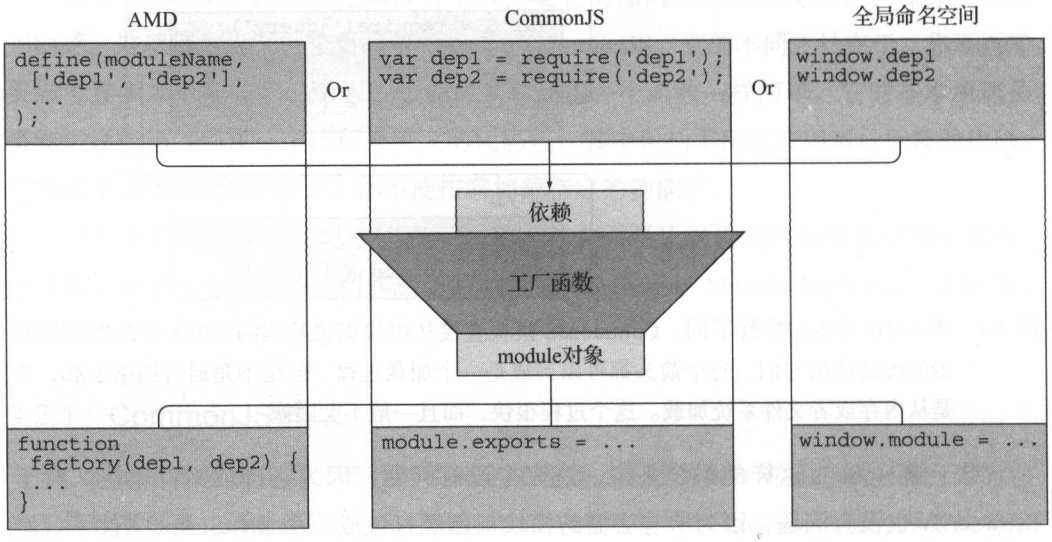



图 9-5 中间的工厂函数是各个模块定义系统的差异所在。根据各个模块定义规范是否能用，有两种方法来调用这个工厂函数：AMD 的方法和 CommonJS 的方法。如果两个都不能用，最后就会把模块添加到 JavaScript 的全局命名空间中

转换成代码，UMD 版本的 toggler 模块就像下面这样：

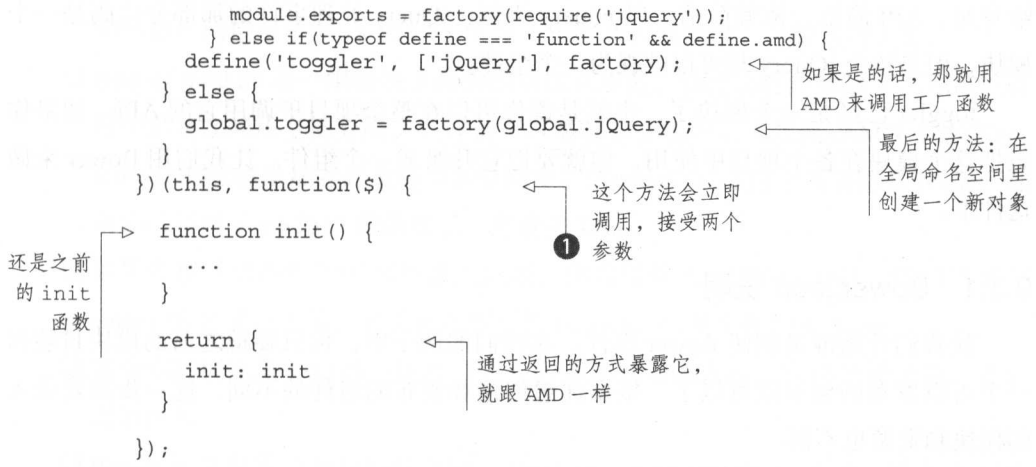
检查是否支持 CommonJS：module 对象和 exports 都应该是可用的

自执行函数接受两个参数：全局命名空间（window 对象或者 Node 的 global 对象）和工厂函数

如果是的话，就用 CommonJS 和工厂函数创建一个对象，然后输出它。这个对象不是这里 require 语句所同步引用的对象

检查 AMD 是否可用：define 方法是否存在，又是否拥有 AMD 属性呢

```
(function (global, factory) {
  if(typeof exports === 'object' && typeof module !== 'undefined') {
```




这里的 this ❶指向的是 window 或者 global 对象。第二个参数是你定义的一个工厂函数。这个工厂函数接受一个参数，就是依赖的模块，在这个例子中是 jQuery。这样依赖，你就可以在任何模块定义规范里使用你的 toggler 模块了，甚至不用模块定义规范都可以！下一步：就是把模块变成一个组件。

### ES2015 的模块

ECMAScript 委员会创造了一个在 JavaScript 中定义模块的标准。它添加了两个新的关键词 export 和 import，尽量独立于各个模块加载器的语法。在本书撰写时，还没有浏览器支持这个新语法。然而，有的工具，比如 rollup.js 已经可以把这种语法编译成 AMD、CommonJS 和 UMD 了。

## 9.3　定义和部署 Bower 组件

模块和组件有共同之处。它们都能把功能封装成自给自足的可复用的单位（unit）。然而，它们的目标却是不同的：模块的目标是创建在代码层面上可维护、可复用的单位。也就是说在项目中，虽然你一直都在不同代码里用模块的功能，但是这些代码只是 API 的调用者。所以说，模块并不在乎文件的组织是怎样的。

另一方面，组件想要在项目架构层面上的复用性。也就是说，一个组件可以由多个模块组成，它的功能也可以比单一的模块更强大。前提是它满足下面这些特点：自

给自足、API 简洁、依赖简洁。以 jQuery 为例：jQuery 处理事件的那部分代码是一个模块，但是整个 jQuery 却可以被视为一个组件。

toggler 已经是一个模块了，也就是说你可以在整个项目里调用它的 API。如果你想把这个模块在各个项目中使用，你就要把它升级成一个组件。让我们用 Bower 来做这件事。

### 9.3.1 Bower.json 说明

让我们开始准备创建 Bower 组件。在我们的例子中，你只要把之前的模块封装到一个可以发布的包里就可以了。根据你想创建和发布的组件的不同，这一步需要引入的模块和资源也不同。

首先，把 toggler.js 放到一个新的空目录中，然后在这个目录里面添加其他必需的文件，把这个孤零零的 toggler.js 变成一个完整的组件。Bower 需要的仅仅是一点元信息，用来把组件添加到组件仓库中。这样其他人就能找到并且安装这个组件了。必要的元信息如下：

- 组件名称
- 组件版本
- 组件的 Git 地址

看起来很熟悉？确实如此。你在安装 Bower 组件或者 NPM 工具的时候就已经用过这三个属性了。元信息保存在组件目录下的一个 JSON 文件里，这个文件和你用来保存依赖的 bower.json 是一样的。你可以回到第 4 章看看安装第一个组件的时候是什么样的。之前，你用它来保存依赖，现在你用它来保存元信息。

首先，在你的包含 JavaScript 模块的新目录中，执行下面的代码。

```
$ bower init
```

但是这一次，你就要认真回答安装提示了。

- name 是 bower.json 唯一必需的属性。随便起一个名字，或者让 Bower 决定。
- description 是组件的简要介绍。它帮助其他人理解这个组件的目标。你可以写简介，也可以省略跳过。
- main file 是一组当你使用 main-bower-files 或者其他工具时所指向的文件，在第 4 章中你已经用过了。强烈推荐设置这个字段。在这个例子中，Bower 建议指向 toggler.js。当然也可以指向更多文件。如果你的组件有多个 main file，或者

你要在代码包里添加 JS 和 CSS 文件，也完全没有问题，尽管加进去吧。

- type of module 是一组模块支持的模块定义规范，你之前已经学过了。你可以在每个名字之间用空格分隔，这样就可以添加多个模块定义规范。AMD 不用多说，globals 的意思是暴露到全局命名空间中。CommonJS 在这个列表里是 Node。如果你已经用了 ES2015 的模块了，那就选 ES6。
- 如果你想让你的组件更容易被人发现，你可以输入一些 keywork，还可以填写 author 的名字、邮箱，定义可选的开源 license，以及组件的 website。
- 如果你已经安装了依赖，你可以把它们添加到依赖表中。在这个例子中，我们之后再加。
- Bower 还会需要你添加忽略规则（common ignore pattern）。通常它会包含那些以点开头的文件（一般是 Unix/Linux 系统中的隐藏文件）、其他 Bower 组件或 node 模块（它们不应该作为组件的一部分，而应该用 bower install 安装），以及测试文件目录。如果你想添加其他要忽略的文件，那就在这里添加。

最后一个问题会问你是否想把它设置为私有的。回答完以后 Bower 会再给你看一次结果，以供确认。如果你接受的话，你就会看到类似下面的结果。

**代码清单 9-5　Bower.json**

```
{
  "name": "toggler",
  "authors": [
    "Stefan Baumgartner"
  ],
  "main": "toggler.js",
  "moduleType": [
    "amd",
    "node",
    "globals"
  ],
  "license": "MIT",
  "ignore": [
    "**/.*",
    "node_modules",
    "bower_components",
    "test",
    "tests"
  ]
}
```

到目前为止，我们还遗漏了一个非常重要的部分：jQuery。你只要通过 Bower 安

装 jQuery，并且加上 --save 参数，它就会被添加到 bower.json 里了。

```
$ bower install --save jquery
```

添加上着最后一部分，你的组件元信息就完整了。不过，组件还是缺少两个重要的部分：版本号和 Git 地址。两者都要用 Git 完成，下一节会详细介绍。

### 9.3.2 部署和发布组件

你还需要两个属性才能把组件发布到 Bower：Git 地址和版本号。Bower 需要通过 Git 来部署组件。Bower 需要一个可以公开访问的 Git 仓库，你可以使用 GitHub、Bitbucket 或者别的什么公共的 Git 托管服务。



注意 Git 这个话题已经大大超出本书的范畴，这一节只会展示基本的步骤。要了解更多有关 Git 和版本控制的信息，推荐阅读 Mike McQuaid 的《Git in Practice》（Manning，2014）等相关书籍。

我们假设你已经在 GitHub 上创建了一个项目仓库，并且已经有了个 URL 用来访问。进入到你组件目录下，初始化本地 Git 仓库：

```
$ git init
```

然后，暂存所有的文件，并且提交：

```
$ git add .
$ git commit -am "initial commit"
```

把 GitHub 上的项目仓库作为远程 Git 仓库添加到本地，然后把所有文件推送到远程：

```
$ git remote add origin https://path-to-your-github-repository
$ git push origin master
```

现在你的文件可以公开访问了，你也有了一个公开的 Git 地址。版本号也可以由 Git 以标签的形式添加：

```
$ git tag -a v1.0.0 -m "version 1.0"
$ git push origin --tags
```

这样做会在 GitHub 上创建一个 1.0 版本。要了解更多的版本号（semantic versioning）和正确版本号的信息，请看附录。有了版本号和 Git 地址，你现在就可以把组件发布到 Bower 仓库了。Bower 仓库是一个可以公开访问的服务器，其中包括了很多组件及其元信息（以 bower.json 的形式），还有对应的 Git 地址。当你通过 Bower 安装一个组件时，

Bower 会在组件仓库中查询组件（根据名字），然后组件仓库会告诉 Bower 到哪里去查找版本和下载文件。图 9-6 展示了这个过程。

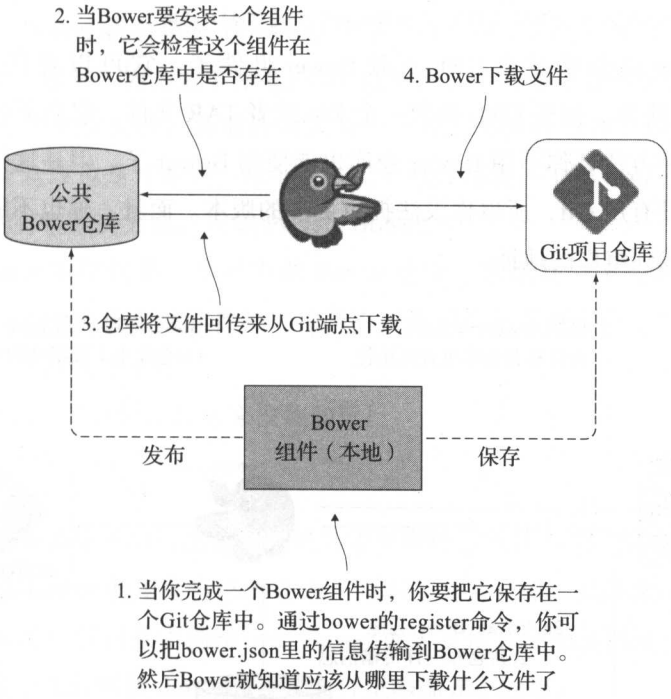



图 9-6　当一个组件准备发布，并且已经保存在 Git 仓库中后，你需要告诉 Bower 仓库到哪里去下载你的组件。Bower 在安装和下载文件的时候，会从 Bower 仓库获取到这些信息

组件部署到 GitHub 上后，你需要告诉 Bower 仓库到哪里去下载你的组件。你可以使用下面的命令：

```
$ bower register <component-name> <git-endpoint>
```

你要把 toggler 和 GitHub 地址填写到命令对应的位置中。然后就完成了！每当你在 GitHub 仓库里部署了新的标签，Bower 都会下载并且安装最新版本的组件。

### 9.3.3　私有项目仓库和组件

Bower 的架构是面向完全的开源的。Bower 仓库里大部分组件也是公开的。即使文档里声称当你把组件上传到 Bower 的公开仓库中后，Bower 组件就应该对所有人公开，也就是说不能放在私有的 Git 仓库中。但是有时候你的组件不适合放在公有的仓库中。你公司的政策可能禁止把组件上传到公网上，或者你的组件里面有一些绝密数据不能给别人看。

Bower对此也提供了解决方案。首先，你不一定要从Bower仓库安装组件。在Bower的安装命令里，你可以在组件名后面添加一个URL作为参数：

```
$ bower install --save http://example.org/main.js
```

这样Bower就会从这个URL下载Bower组件了。它可以是任何文件，CSS、JavaScript都随便你。如果URL指向一个Zip或者TAR文件，它会下载并解压其中的内容。通过这种方法，你不用Bower仓库也能使用Bower了。但是这样也有缺点。首先，由于这里没有用Git，所以你没法拥有更多的版本。而且，你也不能方便地搜索组件了，要自己找到组件的地址。

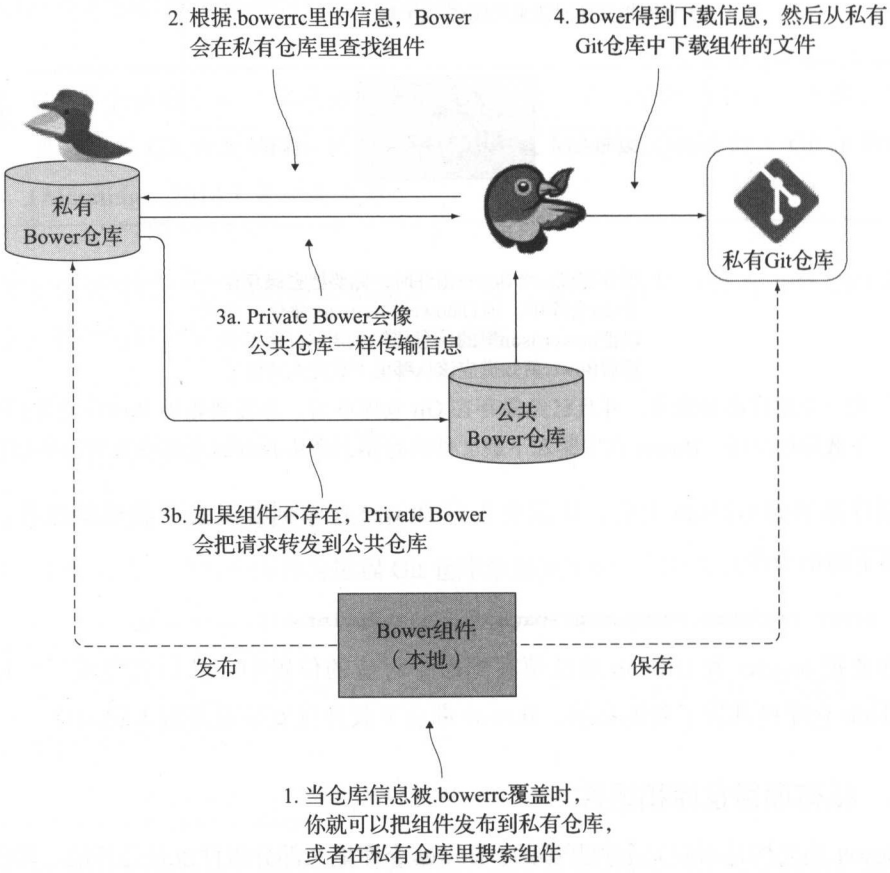



图9-7 一个私有Bower仓库就像一个代理。无论是组件还是项目，要用私有仓库就要在.bowerrc文件里指定组件的地址。私有组件仓库会处理所有Bower的请求，如果组件在私有仓库中不存在的话，还会把请求转发到公共仓库

Bower 不仅允许从 URL 下载组件，还可以从 Git 地址直接下载：

```
$ bower install --save-dev https://github.com/polymerelements/paper-toggle-
    button
```

这样一来，Bower 就会像从 Bower 仓库里一样，从 Git 仓库里下载组件。同时，你也能够拥有多个版本，因为版本标签是 Git 维护的。但是你还是不能搜索组件。

如果你有很多私有组件，你或许可以弄一个私有 Bower 仓库。Bower 是开源的，也就是说 Bower 仓库也是开源的。你可以自己下载并部署一个 Bower 仓库，或者你也可以用更方便的打包解决方案，比如 Private Bower。

Private Bower 不仅仅是一个私有的 Bower 仓库，它也是一个仓库代理。也就是说，Private Bower 首先会看看你要找的组件在私有仓库里是否存在，然后再把请求转发到公共仓库。详情见图 9-7。

Private Bower 可以用 Node.js 来安装和运行：

```
$ npm install -g private-bower
$ private-bower
```

然后这个服务器就可以根据你的主机名或者地址，通过 5678 端口访问了。你要做的事情就是告诉 Bower 使用你自己的仓库，而不是公共仓库。具体做法是，在组件目录和要安装组件的项目目录中创建一个 Bower 配置文件。一个是用来安装的，一个是用来发布的。这个文件叫做 .bowerrc，在文件里添加如下内容。

**代码清单 9-6　.bowerrc**

```
{
  "registry": "http://path.to.your.registry:5678",
  "timeout": 30000
}
```

如果你想在本地尝试 Private Bower 服务器，那就在本地启动服务器，然后访问 http://127.0.0.1:5678，你想怎么玩都可以。

现在，你不仅可以创建模块，还可以创建组件，并且发布到公有或私有仓库中了。

## 9.4　总结

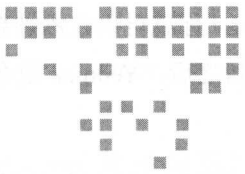
本章内容都是关于模块和组件的：

- 异步模块规范 AMD 可以用来定义以异步形式加载的模块，对于浏览器非常合

适。它有两个方法，define 和 require。它们都要由模块加载的库，比如 require.js 来提供。

- CommonJS 模块以同步的方式工作，它会短暂阻塞执行，需要快速地从内存或文件系统中读取文件。它是为 Node.js 而生的，但是也可以被 Browserify 使用。
- 通用模块规范 UMD 通过工厂函数，结合了 AMD 和 CommonJS 以及全局命名空间这三种定义模块的方式。
- 创建完模块后，我们把模块升级成了组件。我们仔细分析了 bower.json 文件的内容。你通过 Git 部署了组件，并且把它发布到了公共 Bower 仓库上。这样你的组件就能被搜索到，并且能拥有多个版本。
- 如果出于某种原因，你不能公开部署和发布组件，你可以使用 Git 地址、Zip 文件，或者普通的文件来安装组件，不需要组件仓库。但这也意味着你不能享受搜索版本所带来的好处了。
- 如果你有很多私有组件，还有一个选择，那就是使用私有 Bower 仓库或者仓库代理，比如 Private Bower。通过 .bowerrc 文件来把地址指向私有仓库就可以了。

公司的私有 Bower 仓库非常适合用 Yeoman 来创建脚手架。是时候重新回顾 Yeoman 了。



第 10 章 Chapter 10

# Yeoman generator 进阶

**本章包括：**

- sub-generator。
- Yeoman 的文件系统。
- 用户设置。
- generator 组合。

到目前为止，你已经做好了搭建脚手架的准备了。你已经优化了构建工具，加快了开发周期，并且往项目里添加了你自己创建的组件。但是你不可能每天都创建一个新项目。根据项目的大小，你可能要花几周、几个月，甚至几年来扩展、维护、改进老的项目。

即便你的项目一开始都是一样的，随着时间的流逝，它们的差异会越来越大，不仅在内容方面，在架构方面也是如此。比如在项目一开始的时候，你选择了一个模块定义规范（UMD、AMD 或者 CommonJS），每添加一个新模块，项目之间的架构的差异就在扩大。假如你同时开发几个项目，或者有新人加入项目，那么往当前项目里添加新模块就很容易遇到麻烦。

在本章中，你将会学到如何通过改进 Yeoman generator，来克服随着项目发展可能会遇到的摩擦和不一致性。Yeoman generator 可以把项目模板变得很容易复用，这样你和团队的每个成员就能够在各个项目中按照一致的标准来开发了。借助 Yeoman 的

sub-generator，你就可以在添加新项目组件的同时，保持架构的一致性。即便在你已经启动了新项目之后，Yeoman 也会成为你的工具链中的重要部分。

## 10.1 向现有项目中添加新文件

项目进行得越久，里面的文件和模块就越多。尽管你现在的 Yeoman generator 能够在创建新项目时提供一个好的开端，但是在之后扩展代码的时候，它却无能为力。让我们改变这一事实。

在前面的章节中，你已经对模块很熟悉了。模块是封装的、可以复用的代码。在 JavaScript 中，它们要遵循特定的定义规则：AMD、UMD 或者 CommonJS。你开发一个 JavaScript 应用越久，添加的模块就越多。随着时间流逝，你会不断地创建新模块。对于每个模块，你都要做下面的事情：

- 创建一个新文件。
- 往文件里添加模块的模板（参见第 9 章中的 AMD、UMD 或者 CommonJS）。
- 在应用的主文件里引用模块（main.js 或者是 index.html）。

如你所见，这些任务其实都是可以很简单就能自动化的，可以节约大量复制、粘贴的时间，并减少拼写的错误。Yeoman 有一个高级功能就非常适合实现这个需求：sub-generator。你可以用 sub-generator 来向项目中添加新模块或者编辑现有文件。让我们来看看具体要怎么做。

### 10.1.1 sub-generator 的概念

sub-generator 是 Yeoman generator 包的一部分，它做的事情比较小，通常是一个大型、完整的应用 generator 的一部分。它的作用是给应用 generator 提供额外的脚手架，以及给用 Yeoman 生成的项目添加新的元素。比如说你用一个应用 generator 创建的一个新项目（就像第 5 章里一样），这个 generator 已经包含了所有应用必需的文件，而 sub-generator 会给这个项目添加额外的文件和模板，如新的 CommonJS 和 AMD 模块。

从技术上来看，sub-generator 和普通的 generator 结构上类似：它也有模板和组装指令，也能根据内容来填充模板。这一点第 5 章已经讲过了。

它们的区别更多的是概念上的区别。怎么样算一个应用 generator，怎么样算一个 sub-generator 呢？应用 generator 创建的是一个完整的项目；而 sub-generator 的粒度更

细，它只会往已经用应用 generator 创建好的项目里，添加一小部分代码。比如，基于 Browserify/LESS 的 Web 应用的 generator，是一个普通的 generator。而一个往项目里添加 CommonJS 模块的 generator 就是 sub-generator。

sub-generator 的结构也是按照约定的。应用 generator 要在应用目录下通过 yogeneratorname 来调用；sub-generator 的目录和应用目录是同级的，sub-generator 的名字，就是目录的名字。见图 10-1 的示例。

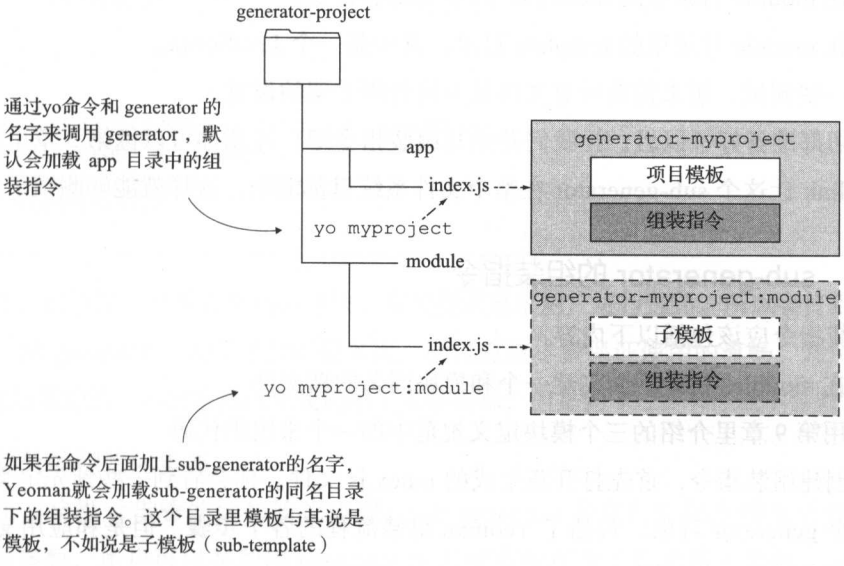



图 10-1　sub-generator 在 Yeoman generator 的中的结构。完整的项目 generator 在 app 目录下。sub-generator 目录与它同级，并且以 sub-generator 的名字为名，其中是它自己的组装指令和模板

正如图 10-1 所示，你可以通过在命令后面加上一个冒号和 sub-generator 的名字来单独调用它。在本例中，它的名字是 module，因为它的作用就是往项目里添加新模块。Yeoman 接下来默认会调用目录中的组装指令，让我们创建一个 sub-generator 吧。

在第 5 章中，你通过 generator-generator 包来创建了一个新 generator。你猜怎么着？这个 generator 还有一个用来创建 sub-generator 的 sub-generator！回到第 5 章中的 myproject generator 目录，执行下面的命令：

```
$ yo generator:subgenerator module
```

这个命令的结构是这样的：

- yo generator 会调用名为 generator-generator 的 generator。
- :subgenerator 指向的是 generator-generator 中的 subgenertator 目录。
- module 是你要创建的 sub-generator 的名字。

执行完命令后，你会得到下面这些文件：

- 在 generator 目录下的一个名为 module 的目录。
- 在 module 目录里的 index.js，其中是组装指令。
- 在 module 目录里的 template 目录，其中是一个 JavaScript。
- 一些测试，用来检查所有文件是否符合脚手架的设置。

一切都准备好了以后，让我们开始写组装指令吧！注意你可以像第 5 章里面那样，用 npm link 让这个 sub-generator 在整个文件系统里都能用，这样就能够测试了。

### 10.1.2 sub-generator 的组装指令

组装指令应该包括以下内容：

- 在 modules 目录下面创建一个和模块同名的新文件
- 用第 9 章里介绍的三个模块定义规范中的一个来组织代码

要创建组装指令，首先打开新生成的 index.js 文件。你会看到一些预先定义好的函数和一个 generator 对象，包括了 Yeoman 组装流程的各个步骤。但是和应用 generator 相比，你会发现一些不同。

#### 1. 没有安装步骤

sub-generator 应该只往项目里添加小部分代码（最好是用同一个应用 generator 搭建的项目）。因此，这里不会发生安装依赖这种大型操作，这也就意味着相比于完整的应用 generator，sub-generator 省略了安装步骤。只有在你真正需要依赖的时候，才应该加上安装步骤。

#### 2. 没有安装提示步骤

sub-generator 应该尽可能少用安装提示（就是询问详细信息的问卷）。因此，sub-generator 默认不包含安装提示步骤。然而在本例中，你还是需要一个安装提示步骤，来让用户选择 AMD、UMD 还是 CommonJS。所以你要自己添加安装提示步骤。下面的代码清单会告诉你如何添加 prompting 方法。

**代码清单 10-1　module/index.js 中的 prompting() 函数**

```
prompting: function () {
  var done = this.async();

  var prompts = [{
    type: 'list',
    name: 'moduletype',
    message: 'Which module type do you want to include',
    choices: [
      'UMD',
      'AMD',
      'CommonJS'
    ],
    default: 'UMD'
  }];

  this.prompt(prompts, function (props) {
    this.props = props;
    done();
  }.bind(this));
},
```

因为是从列表中选择，所以要选择 list 类型

把结果保存在 moduletype 属性中

记住，安装提示是没有办法的办法。安装提示要求用户进行选择，所以会打断操作的流畅性。sub-generator 一天可能会用很多次，所以应该尽量保证使用的流畅。调用它，就完成，这是最好的。不过虽然这里我们用了安装提示，但是到 10.2.2 节你就不会被打断了。

### 3. 额外的初始化步骤

取代安装提示步骤的是初始化步骤。sub-generator 会在初始化步骤来处理命令行中传入的参数，用户可以在调用 generator 的时候直接在命令行里传入参数，而不用回答一堆问卷。创建好的 sub-generator 已经有一个参数了，你可以在下面的代码中看到。

**代码清单 10-2　module/index.js 节选**

```
initializing: function () {
  this.argument('name', {
    required: true,
    type: String,
    desc: 'The module name'
  });
},
…
```

❶ 通过调用 this.argument 来定义命令行参数

❷ 参数的具体信息则用一个配置对象定义

在本例中，我们想保存新创建的模块的名称❶。你可以把它设置成 required（如果不填这个参数，Yeoman generator 会崩溃），然后选择一个 JavaScript 类型，并填写描述❷。

可以这样来调用：

```
$ yo mygenerator:module newModuleName
```

这样就能把 name 参数设置成 newModuleName。当你通过 generator:subgenerator 创建 sub-generator 的时候，代码清单 10-2 里的代码就已经在文件里面写好了。你不需要做任何修改，因为在本例中，我们正好需要 name 参数。下一节你应该很熟悉。

#### 4. 写入文件步骤

写入文件步骤和应用 generator 是一样的。基于来自上一步的输入，你要把文件从模板目录中复制到项目目录中。基于在安装提示步骤中用户的选择，你要从模块模板中选择其中一个，然后根据它和初始化步骤的参数来创建文件。

> **GitHub 上的示例模板** 请在 GitHub 上的 template 目录中获取必要的文件，链接是 https://github.com/frontend-tooling/sub-generator-template，然后把 JavaScript 文件复制到你的 templates 目录中。

下面是具体代码。

**代码清单 10-3 module/index.js 中的写入文件步骤**

```
writing: function () {
  var fromFile = this.props.moduletype.toLowerCase();
  var toFile = this.name;
  this.props.moduleName = this.name;
  this.fs.copyTpl(
    this.templatePath(fromFile + '.js'),
    this.destinationPath('modules/' + toFile + '.js'),
    this.props
  );
}
```

模块模板的名字和代码清单 10-1 里一样；使用 toLowerCase 方法确保文件系统能找到正确的模板文件

在 props 字段里填写模块名字

这个对象会被传递给 copyYpl 函数，然后选择的模板就会被复制到目标目录中

新模块的名字以初始化阶段传入的参数命名

只要写这么几行代码，就可以随便添加新模块了。下一节将进一步优化这个 generator，让它变得更好用。

## 10.2 改进接口

现在我们已经可以根据模板来创建新模块了，这很不错，但是还可以更加自动化：

- 新模块还是要手动在 JavaScript 主文件里添加，否则就不会被加载。为什么不自动把模块添加到主文件里呢？
- 一般来说，你不会在一个项目里混用模块定义规范。当你已经决定要用一个模块定义规范后，你就会在整个项目里沿用。所以你可以把选择模块定义规范的步骤自动化，让它自动选择第一次的选择。

加上这两步，generator 的用户接口就更加精简了，往项目里添加新模块的步骤也减少了。让我们来实现它吧！

### 10.2.1　修改现有的文件

每次你创建一个新模块的时候，你都要把它引入到 JavaScript 文件中。你既可以在其他用到它的模块中引用它，也可以在一个主文件里引用所有其他模块。让我们以 CommonJS 为例，比如你想把新创建的模块添加到 main.js 的 require 声明中。实际的 main.js 代码应该是这样的。

**代码清单 10-4　main.js**

```
var jquery = require('jquery');
var moduleX = require('./modules/moduleX');
// 添加更多的模块
```

所有新创建的模块都可以添加到第三行注释上面，添加的过程可能会很枯燥无聊。打开 main.js 文件，把光标移到正确的位置，输入正确的路径，不要拼写错误……看起来这些事情都可以轻松地用 Yeoman 来自动化。具体要自动化的步骤是，打开 main.js 文件，添加几行代码，保存到硬盘。你可以使用 Node.js 文件系统的 API 或者使用 Yeoman 提供的更方便的 API 来做这件事。Yeoman 有一个叫 fs 的文件系统的代码库。在之前复制模板的时候已经用过了。但是这个文件系统代码库还有很多其他功能，比如从硬盘中读写文件：

- this.fs.read 接受一个文件系统的路径，然后返回一个字符串，其中就是文件内容。这个函数是同步的，所以比起 Node.js 中对应的读文件的 API，它用起来更加容易控制。this.fs.readJSON 方法的作用是一样的，但是它读取的是一个 JSON 文件，返回的是一个 JavaScript 对象。
- this.fs.write 也接受一个文件系统路径作为参数，并且把一个字符串保存到文件中。它会覆盖文件的内容，但是事先会询问是否要覆盖。this.fs.writeJSON 的作

用也类似，但是是把 JavaScript 对象保存为 JSON 文件的。

让我们尝试用一下上面的函数。下面的代码展示了如何添加一个新的 CommonJS 模块的 require 声明。

**代码清单 10-5 module/index.js 写入步骤**

```
writing: function () {
  var moduleType = this.props.moduletype.toLowerCase();
  var toFile = this.name;
  this.props.moduleName = this.name;
  this.fs.copyTpl(
    this.templatePath(moduleType + '.js'),
    this.destinationPath('modules/' + toFile + '.js'),
    this.props
  );
  if(moduleType === 'commonjs') {
    var destPath = this.destinationPath('main.js');
    var contents = this.fs.read(destPath);
    var index = contents.indexOf('// Add more modules');
    contents = contents.substring(0, index) +
      "var " + toFile + " = require('./modules/" + toFile + "')\n"
      + contents.substring(index, contents.length);
    this.fs.write(destPath, contents);
  }
}
```

destinationPath 中的文件是调用 generator 的目录中的文件，这里是 main.js

读取文件中的所有内容

这个 require 声明需要添加在注释那行（本例中是第三行）之前，所以要找到这行注释起始的位置

前面添加一行声明；JavaScript 的 substring 方法在这里很有用

把内容保存到文件中 ❶

这样做会覆盖整个 main.js 的内容❶。为了确保不出意外，所以需要用户确认是否覆盖。这就需要一些提示。

现在你不仅会往项目里添加新模块，而且还会加载它们了。你可以把这套方法也用在 UMD 和 AMD 的模块上。这样一来你的用户就可以立即开始写代码，不用管其他代码组织结构方面的事情了。只要一行命令，一切就都搞定。

## 10.2.2 处理用户的配置

在你的模块的 sub-generator，你要添加一个提示步骤，让用户选择使用哪个模块定义系统。这会打断整个流程，因为用户每次创建一个新模块的时候都要选择一次。这样一来每天可能都要选好几次。但是一般而言，用户一旦选择过一次，之后就不会轻易改变了。为什么不默认选择他们之前选择的选项呢？

Yeoman 有一个用来做这件事的技术叫*用户配置*（user configuration）。它是一个 JSON 文件，每个项目分别都有一个。generator 可以读取它，来获得用户配置。它也是

可写的，所以你在里面保存多少配置参数都没问题，用户配置文件的名字叫作 .yo-rc.json，内容大概是下面的样子。

**代码清单 10-6　.yo-rc.json 示例**

```
{
  "generator-myproject": {
     "moduletype": "amd"
  }
}
```

❶ Yeoman 的资源文件都是 JSON 文件。其中的 JSON 的第一层级是项目中用到的 generator 的列表

❷ 第二层级是这个 generator 的配置

在之前的例子中，你只用到了 myproject generator ❶。在这个例子中，我们预先选择了要使用的 generator ❷。

这个文件用 Yeoman 提供的 API 来读写。让我们看一个直观的例子。设想一下之前的提示场景。你要让用户选择他们想要的模块定义系统。当用户选择好以后，把选项保存起来，详情参见图 10-2。

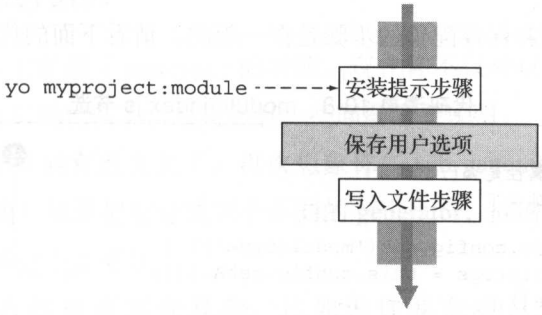



图 10-2　你可以在安装提示步骤和写入步骤和写入文件步骤之间再加一个步骤，用来保存用户的选项

下面的代码展示了如何用 Yeoman 的 API 来实现。

**代码清单 10-7　module/index.js 节选**

```
...
this.prompt(prompts, function (props) {
  this.props = props;
  this.config.set(props);
  done();
}.bind(this));
...
```

❶ config 对象是在当前 Yeoman generator 中的

通过 set 方法你可以把整个属性和值的对象传进去❶。当然你也可以一个一个地设置属性和值。正如代码所示，你可以使用 this.config.set 接口把配置的属性传递到一个

文件中，这样你的配置就保存好了。

让我们进入下一个模块。你保存了配置（无论是 AMD、UMD 还是 CommonJS），然后想在创建新模块的时候默认选择这个配置。为此，你需要在安装提示步骤前创建一个新的步骤，用来检查配置。如果已经有配置了，那么就可以直接写入文件了。图 10-3 可视化了这个流程。

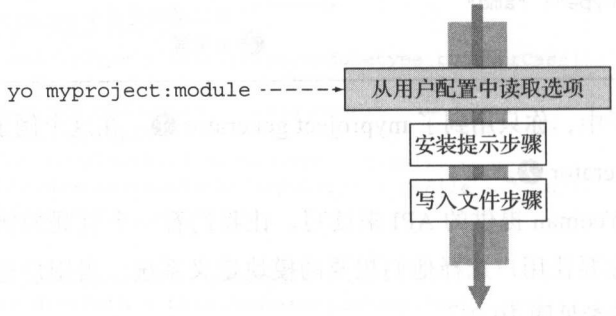



图 10-3 在下一个模块中，你要在安装提示之前，加载用户配置

获取配置的步骤和保存配置的步骤是在一起的。请看下面的代码。

**代码清单 10-8 module/index.js 节选**

```
检查要提示的属性是否已经
保存在配置文件中了
  if(this.config.get('moduletype')) {
    this.props = this.config.getAll();
    done();
  } else {
    this.prompt(prompts, function (props) {
      this.props = props;
      this.config.set(props);
      done();
    }.bind(this));
  }
```

❶ 如果是，那就读取配置文件中的属性，然后保存在 props 对象中

调用 done ❷

如果不是，那就提示该提示的属性

这里模拟了之前安装提示的作用，我们把保存好的属性放到 Yeoman generator 中❶。这一步是异步的，通常需要回调，但是你可以在同步的代码中调用 done 来表示函数已经完成了❷。

在本例中，你使用了 get 方法来检查特定的属性，还使用了 getAll 方法来加载所有的属性。最后一个要介绍的方法是 delete，通过它你可以删除某个已经保存了的属性。你可以在初始化或者重置 generator 的时候使用这个方法，如下代码所示。

**代码清单 10-9　module/index.js——初始化步骤**

```
this.option('reset');                          ❶ 定义一个 reset 选项
if(this.options.reset) {                       检查是否传入了 reset
  this.config.delete('moduletype');            选项 ❷
}
```

这样我们就可以在命令行中给 generator 传入 --reset 选项了❶。传入了这个选项，就可以删除已经保存了的 moduletype 属性。只要这些代码，你就可以这样调用你的 generator：yo myproject:module moduleName--reset。这么做可以重置用户配置，并且创建一个新的名为 moduleName 的模块。

有了用户配置后，generator 的工作效率更高了。每当你创建新模块的时候，Yeoman 就已经知道该用哪种模块定义系统了，从而不需要打断流程询问用户，就可以创建出这个模块。

## 10.3　组合 generator

sub-generator 大大扩展了 generator 的功能。现在你不仅可以创建一个新项目，还可以创建新模块或者扩展代码了。但是还有一些问题需要说明：

- sub-generator 可能有点太大了。再次想象你的 sub-generator，它要处理三种不同类型的模块！如果把它分成三个不同的 generator，同时使用一个接口，分离关注点，这样会不会更好？
- generator 会有些过度冗余复杂。比如说你要在项目里用流行的 UI 框架 Bootstrap。现在已经有了一个现成的 generator-bootstrap，但是它给用户提供的选项太多了，比如选择预处理器。你想要让它按照你的项目标准执行，所以最后你可能得自己创建一个 generator。

Yeoman 提供了一个功能来解决这两个问题：generator 组合（generator composition）。它能够让你在一个 generator 中调用另一个 generator，并且还能传递选项和参数。让我们来看一看。

### 10.3.1　全局的 generator 的组合

让我们首先来解决第一个问题。你想要把 AMD、UMD 和 CommonJS 分成若干个 sub-generator，然后使用 myproject:module generator 来作为它们三个的共同的接口。这

样就可以做到关注点分离（separation of concerns）。你可以用 sub-generator 做不同的事情，然后在需要的时候单独调用它们。module sub-generator 仍然会帮你默认选择模块定义系统，但是它不再负责模板文件了。模板文件都会被移到其他 generator 中。图 10-4 展示了应该怎么做。

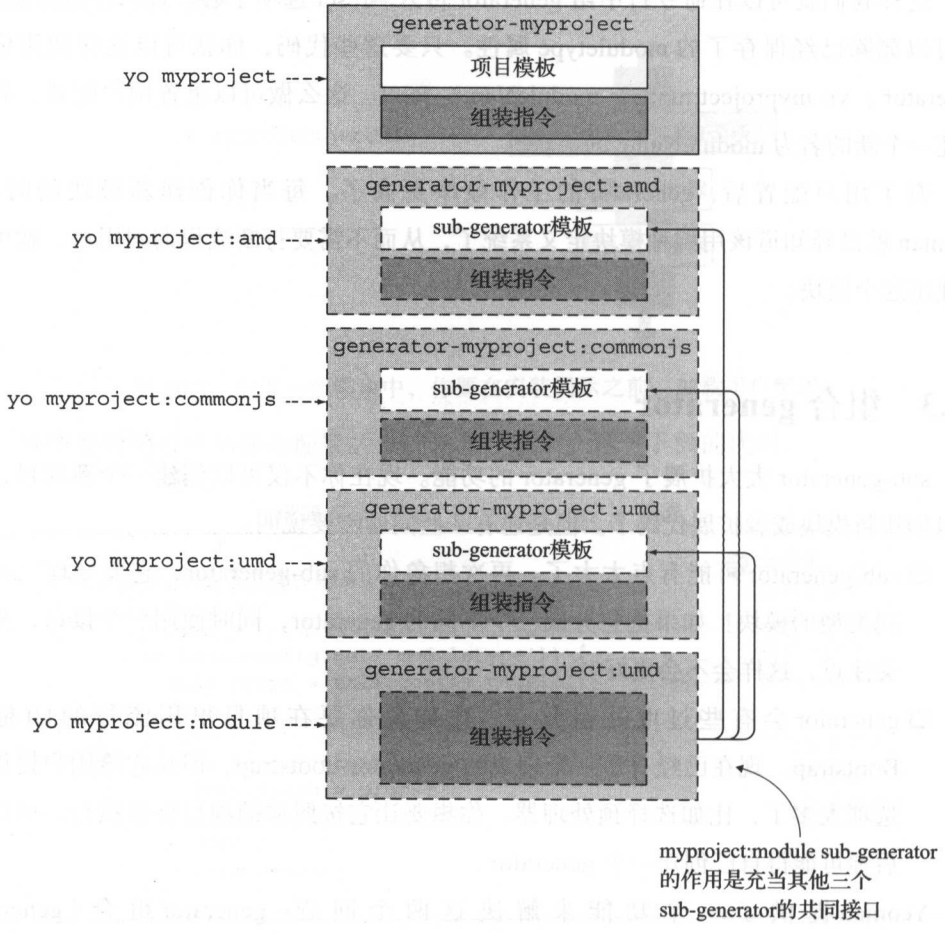



图 10-4 generator 组合可以让你在 generator 中使用其他 generator。这样可以快速地调用其他 generator（比如让 myproject:module 自动跳转到其他某个 module generator），并且可以把写入文件步骤分离出来。途中每一行都展示了调用不同 generator 所用的不同的命令。但是通过 module generator，你就可以用一个命令来调用其他 sub-generator 了（AMD、UMD 或者 CommonJS）

让我们把它转换成代码吧。首先，你要把 module generator 重构成三个看起来差不

多的 sub-generator，分别对应 AMD、UMD 和 CommonJS，如下代码所示。

**代码清单 10-10　amd/index.js**

```
'use strict';
var yeoman = require('yeoman-generator');

module.exports = yeoman.generators.Base.extend({
  initializing: function () {
    this.argument('name', {
      required: true,
      type: String,
      desc: 'The AMD name'
    });
    this.log('You called the AMD with the argument ' + this.name + '.');
  },

  writing: function () {
    var toFile = this.name;
    this.props = {};
    this.props.moduleName = this.name;
    this.fs.copyTpl(
      this.templatePath('amd.js'),
      this.destinationPath('modules/' + toFile + '.js'),
      this.props
    );
  }
});
```

❶ AMD generator 要跳过安装提示步骤，因为这里不需要选择

填上唯一的一个参数——名字

然后把模板复制到目标目录 ❷

我们想创建 AMD 模块❶，这样就可以了！虽然我们跳过了安装提示步骤，但是最后还是要传入一个 props，所以我们自己创建一个。UMD 和 CommonJS 的模块的代码也和这个类似❷。

如此一来，你就有了三个分离的 generator，可以各自被调用。它们都没有安装提示步骤，sub-generator 就应该这个样子。

module generator 需要一个新的写入文件步骤。在这个步骤里，你只要选择正确的 sub-generator 就可以了，代码如下。由于 myproject generator 是全局安装的，所以 Yeoman 知道要到哪里去找你想调用的 sub-generator。

**代码清单 10-11　module/index.js——写入文件步骤**

```
...
writing: function () {
  var name = this.name;
  var module = this.props.moduletype.toLowerCase();
  this.composeWith('myproject:' + module, {
    args: [ name ]
  })
}
...
```

这里并不选择文件，而是调用相应的 sub-generator

用 composeWith 调用 generator。它接受两个参数：generator 的名字和具有 option 和 argument 的一个对象 ❶

这里我们把命令行参数传入的名称传递给另一个 generator ❶。当你组合 generator 时，你要传递两个参数：option 和 argument。

argument 是一个数组。它包括了一系列从命令行传入到 generator 的值。在本例中，也就是模块的名称。option 是一个对象。它包括一系列属性和值，这些属性和值将会被 Yeoman 解释为命令行选项（比如 reset 这个 flag）。

现在你的用户有了一个接口，根据用户的选择，他们将会调用三个 sub-generator 中的一个。但是可以根据自己的需求扩展它们各自的写入文件步骤。

### 10.3.2 依赖和 generator 的组合

回到 Bootstrap 的例子，在项目的 generator 里用 Bootstrap。你不想让用户在安装 Bootstrap 的时候选择 LESS 以外的预处理器，因为你的项目就是基于 LESS 的。但是你所使用的原始的 generator-bootstrap 是可以让用户选择预处理器的，该怎么办呢？是你自己重新创造一个安装 Bootstrap 的方法，还是利用 generator 组合的能力来预先选好一个预处理器呢？如果你选择自己创造一个方法，那就意味着你要自己维护这个项目。但如果你在 generator 里用原始的 Bootstrap generator，那就意味着 Bootstrap generator 团队会帮你维护这个项目。如果让我选的话，肯定是选后者。

一开始的步骤和之前一样。不同的是，这次你加载的不是一个全局 generator，而是在全局 node 模块目录里的 generator。你要从那里面安装一个 generator 作为依赖，并且加载它，见图 10-5。

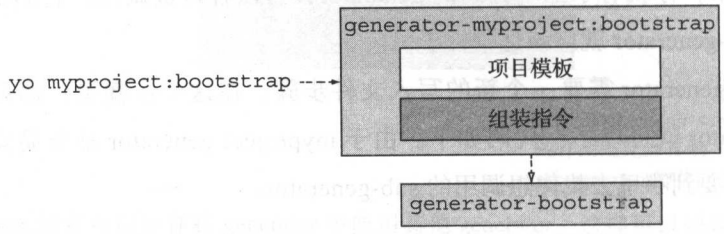



图 10-5 generator-bootstrap 是作为 generator 的依赖引入的，所以它在 node_modules 目录中。通过解析这个 generator 的路径，就可以在你自己的 generator 中，调用它和它的 sub-generator 了，并不一定需要全局安装

为了实现它，首先要在本地安装 generator-bootstrap：

```
$ npm install -save generator-boostrap
```

这样做，Bootstrap generator 就会和你自己的 generator 安装在一起，但是只有

generator-myproject 本地才能访问，而不是全局的。这也就是为什么你必须要在传递选项的时候填上路径。见下面的代码。

**代码清单 10-12　bootstrap/index.js**

```
'use strict';
var yeoman = require('yeoman-generator');

module.exports = yeoman.generators.Base.extend({

  writing: function () {
    this.composeWith('bootstrap', {
      options: {
        format: 'less'
      }
    }, {
      local: require.resolve('generator-bootstrap/app/index.js')
    });
  }
});
```

bootstrap sub-generator 有一个写入文件步骤，在这个步骤里你要通过 composeWith 函数来调用 generator-bootstrap

把选项填进去，这样用户就不能选择其他预处理器了 ❶

你不能指望用户去全局安装 generator-bootstrap，所以你要把它作为依赖在本地安装 ❷

在本例中，我们选择 less ❶。你可以查看 generator-bootstrap 的文档来决定哪些参数可以调用。这需要 Yeoman 知道在哪里找到 generator 文件❷。因此，你要用 require.resolve 来找到正确的 generator 文件的路径。

这样就会跳过 generator-bootstrap 的安装提示步骤，并且预先选择 less。你还可以扩展。generator-bootstrap 做的事情不仅仅是安装一些 Bower 组件而已，你还可以在 main.less 里面引用 Bootstrap 的组件的代码，代码如下。

**代码清单 10-13　bootstrap/index.js**

```
'use strict';
var yeoman = require('yeoman-generator');

module.exports = yeoman.generators.Base.extend({

  writing: function () {
    this.composeWith('bootstrap', {
      options: {
        format: 'less'
      }
    }, {
      local: require.resolve('generator-bootstrap/app/index.js');
    });

      var dpath = this.destinationPath('styles/main.less');
      var contents = this.fs.read(dpath);
      contents += '\n@import "../bower_components/"' +
```

❶ 通过 Yeoman 的 API，你可以很容易地找到 main.less 文件

通过 read 方法来加载这个文件的内容

这个方法会返回一个字符串，所以你可以直接添加一行代码 ❷

```
            'components-bootstrap/less/bootstrap";'
        this.fs.write(dpath, contents);          <-- 把内容重新写入文件系统
    }
});
```

这段代码是用应用 generator 创建的❶。在本例中，要把新增的代码添加到文件最后❷。

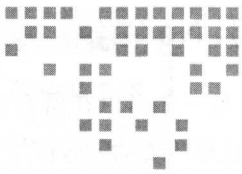
完成这最后一步后，你的 generator 已经像瑞士军刀一样强大了：你可以用同一个接口，调用不同的 module generator，来帮助用户更好地管理他的项目。

## 10.4 总结

本书第二部分的主要内容是复用构建和开发的流程，而最后一章则是关于 Yeoman 在其中的作用：

- sub-generator 让你可以修改搭建好的项目。你可以往项目里添加新的模块，而这个过程完全是自动完成的。你还实现了 UMD、AMD 和 CommonJS 模块的 sub-generator。
- Yeoman 文件系统 API 在从硬盘读取文本和 JSON 文件时非常好用。你可以使用它，并且加上是否要覆盖现有文件的提示，来修改现有的文件。这在添加新模块到主文件中的时候非常方便。
- 用户配置能够保存用户的选项，这可以带来很多好处。你可以预先选择好选项，加快脚手架搭建的速度；还可以保存用户的配置，供下一次添加新模块的时候使用。Yeoman 的用户配置 API 在加载和保存配置的时候也很方便。
- generator 组合让你可以使用其他（sub-）generator，来避免代码的重复。通过组合 generator，你还可以对现有的 generator 进行修改，来省去那些原本需要用户输入的参数。

通过 sub-generator 和它们的 API，你创建了一个强大的工具，它会给你的开发带来很大的便利。Yeoman 也成为了你的工具链的重要的一部分。未来不仅是在启动项目时会用到 Yeoman，在项目开发过程中同样也会不断用到。



附录 A Appendix A

# Node.js 导论

**本章包括：**

- 基于 JavaScript 的工具的 Node.js 生态系统。
- 使用 NPM 安装新的基于 Node.js 的模块。
- 本地安装和全局安装的区别。
- package.json 文件的内容。
- 处理版本号的正确方法。

2009 年，Joyent 公司的一群聪明人发布了 Node.js——一个对 Google V8 JavaScript 引擎的独立的实现。它的最初目标是实现一个供高速网络应用运行的环境。Joyent 没想到的是，在第一次宣传的多年以后，开发者们会用 Node.js 做各种各样的事情：服务器、命令行应用、甚至搭载网络摄像头的无人机。

Node.js 之所以这么流行，主要还是因为它可以在服务端运行 JavaScript 这个最基本的功能，但是 Node.js 能做的事情远不止这些。它打破了浏览器环境的限制（浏览器仅仅是用来展示的），提供了访问操作系统的能力，而这一切所用的又是前端开发者们最熟悉的语言。正是这种可以用同一种变成语言实现几乎任何目标的能力，造就了 Node.js 的成功。

附录的主要目的是通过例子给读者一个对于 Node.js 及其生态系统的大概的了解。

## A.1 初识Node.js

Node.js是一个可以在所有流行操作系统上运行的平台，包括Linux、Windows和Mac OSX。这里所说的“运行”，并不是说它在所有平台上的表现都是一样的。事实上，在不同操作系统中，Node.js也会有点区别，主要是在读取文件的时候，但是大部分时候开发者是永远也不会遇到这方面的问题的。

作为阅读本章的前提，读者应该熟悉自己所用的操作系统的命令行接口（Mac上的终端，Windows的命令提示符和Linux的Shell）。你不需要非常精通，但是应该要知道如何访问文件系统，如何执行简单地命令[⊖]。

准备好了吗？我们开始吧！

### A.1.1 安装Node.js

安装Node.js的方法有很多，最简单地一种是打开它的官网[⊜]，下载对应你的操作系统的安装包，然后按照向导安装就可以了。Windows和Mac OSX都可以用这种方法安装。

在Linux上，你可以通过各个发行版的包管理器安装。用这种方法安装的Node.js都能完美适合你的操作系统环境，比从网上安装要好很多。

在Windows安装时，一定要确保勾选图A-1中展示的选项，把Node和NPM添加到PATH环境变量中。NPM很重要，因为你要通过NPM才能安装Node.js模块。这样一来，你就可以用命令行接口来使用这些工具，无论你用的是什么操作系统。如果你安装的时候忘记勾选了，你可以重新安装一次。

安装完毕后，你要检查Node.js命令是否真的可以用CLI（命令行接口）访问了。打开你的CLI，输入下面命令：

```
$ node --version
```

如果一切正常，你就会看到Node.js的版本信息。如果你输入：

```
$ node
```

---

⊖ Mac/Linux的命令行接口的参考资料：http://cli.learncodethehardway.org/book/。Windows的参考资料：http://www.cs.princeton.edu/courses/archive/spr05/cos126/cmdprompt.html。

⊜ http://nodejs.org。

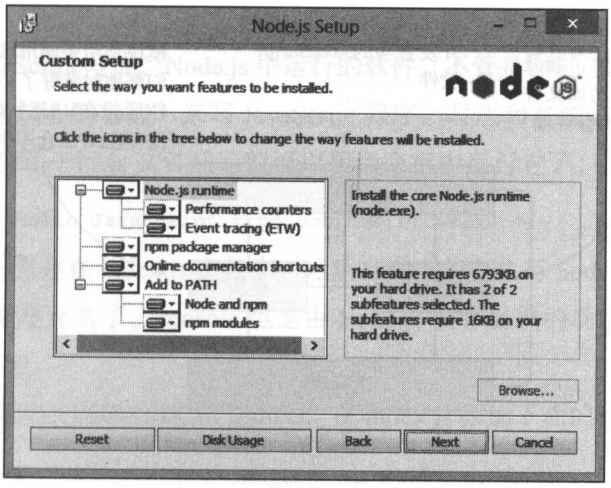



图 A-1　必须要勾选图中的选项，才能正确配置好环境

那么你就会进入到一个和浏览器开发者工具的控制台类似的界面，你可以在里面执行 JavaScript 代码，如图 A-2 所示。在 Windows 中，你还会在开始菜单中找到一个打开这个控制台的快捷方式。

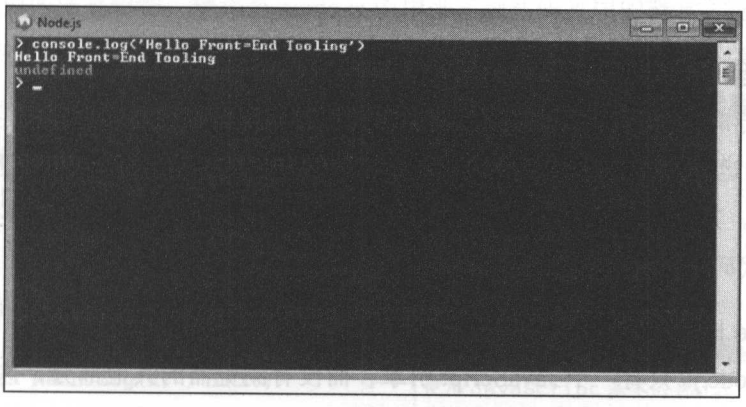



图 A-2　Node.js 控制台中的 JavaScript

## A.1.2　Node.js 生态系统

如果你注意观察安装步骤，你会发现 Node.js 不仅仅只是它自己组成的。相反，除了 Node.js 运行时以外，你还安装了其他一些组件。所有这些组件共同构成了 Node.js 的生态系统，如图 A-3 所示。

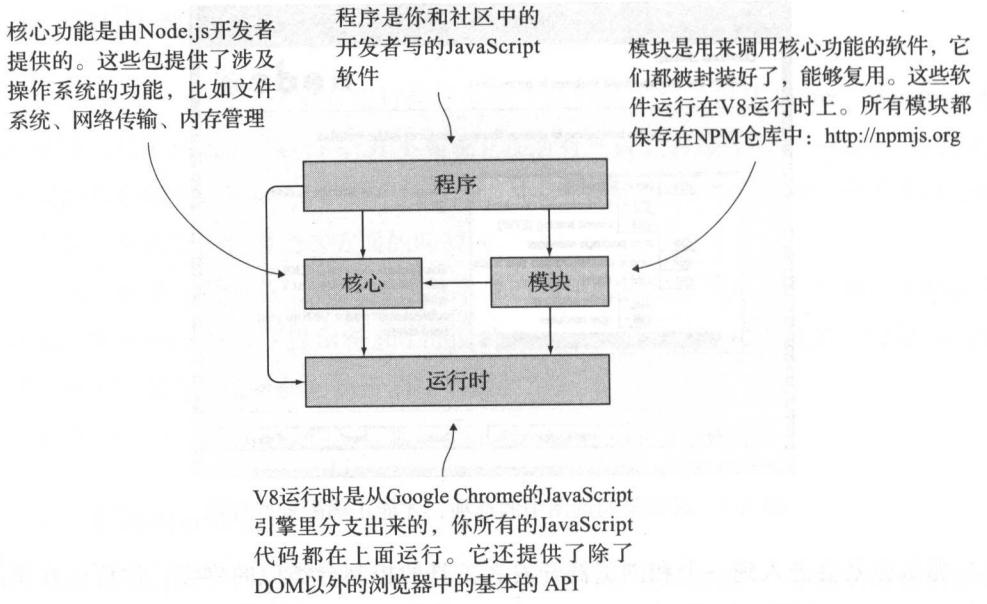



图 A-3 Node.js 生态系统。它的基础是 V8 运行时。V8 运行时是从 Google Chrome JavaScript 引擎分支出来的。Node.js 的开发者提供了核心功能，用来访问操作系统。最上层的是程序和模块

### 1. V8 运行时

V8 运行时是从 Google Chrome JavaScript 引擎分支出来的，它不需要任何浏览器环境就能运行。这也就意味着，你不能在里面做任何涉及浏览器环境的事情（比如，DOM）。其中也没有 navigator 和 window 对象。它只能用来处理基本语法，所有 JavaScript 代码都会在这个运行时中编译和运行。

### 2. Node.js 核心

既然 Node.js 只有一个 JavaScript 引擎，而没有浏览器的其他部分，你或许就会猜想 Node.js 没有任何浏览器的 API。的确如此，所有和 HTML 相关的功能都被剥离了。但是要在 Node.js 中开发软件，你还是需要一些基本的 API。Node.js 核心就能提供这些 API，而且是专门在服务端运行的。这些 API 涉及文件系统、网络协议和流。总而言之，所有关于输入输出以及操作系统的功能，在 Node.js 中应有尽有。Node.js 被设计得尽可能轻量，这也就意味着只有那些完全无法通过 JavaScript 语言本身实现的功能才会包括在其中。

### 3. 程序

程序是用 JavaScript 写成、Node.js 中运行的软件，不管是你自己写的还是第三方提供的。你可以引用核心的功能，来写 JavaScript 程序，写法和在浏览器中写是一样的。把所有代码都打包到一个 .js 文件后，就可以用命令行中运行它了，下面是一个例子。

```
console.log("Hello JavaScript from not the browser! ");
```

把上面的代码保存在 index.js 文件中，然后在命令行里执行 node index.js 命令。你大概能够猜到结果是什么了。console 也是由 Node.js 核心提供的功能，其余的任务则是由 V8 引擎完成的。

在类 unix 系统中（比如 OSX 和 Linux），在 index.js 添加下面的代码。

```
#! /usr/bin/env node
```

这样就可以直接执行这个文件，而不用把它当作一个参数传递给 node 命令了。要在 Windows 上实现同样效果，你需要把它封装一下。封装的工具也是由 NPM 自动提供的（之后会详细介绍），这样你就可以创建自己的命令行工具了。

### 4. 模块

最后要说的就是模块。模块是 Node.js 中用来实现代码复用的概念，你写的每个程序都可以被打包成一个模块用来发布。模块所包含的封装好的功能，可以在新程序中复用。每个 JavaScript 文件都可以被视为一个模块。要把你的模块的功能提供给其他程序，你要通过一个名叫 module 的全局对象来输出函数和 JavaScript 对象。通过 exports 属性，你可以定义对外暴露的函数或者对象，如下面的代码所示。

```
var LN10 = 2.302585092994046;          <-- 定义一个常亮，保存 ln10 的值

module.exports = function(x) {         <-- module.exports 是 Node 特有的
  return Math.log(x) / LN10;           <-- 使用 JavaScript 基本的 Math 库的功能（不需要加载）
}
```

把上面的代码保存到 log10.js 文件中，你就有了一个函数可以用来快速地进行对数计算。要使用这个函数，你要在其他模块或者程序中 require 它。当你 require 一个模块时（比如通过文件名），实际上你就是加载了通过 module.exports 所暴露出来的功能。

```
var log10 = require('./log10');          <-- require 和 module.exports 是相对的。
console.log(log10(Math.random());        <-- 你可以在你的软件中使用这个功能。
```

**Node模块** 注意，上面的代码在require的时候，并没有在文件名后添加.js的后缀。模块可以通过NPM（全球的模块仓库）来发布，也可以通过SVN或Git等版本控制系统发布。

## A.2 通过NPM安装全局和本地的模块

当你安装完Node后，你不仅安装了整个核心和运行时，而且还安装了NPM——从中心服务器安装Node模块的软件。NPM可能是“Node Package Manager”的首字母缩写，但是也有开发者开玩笑说它不是首字母缩写[⊖]。我们还是姑且把它当作是Node的包管理器，因为它的功能事实上也是如此。

### A.2.1 全局安装和本地安装

在Node.js中，有两种安装模块的方式：全局和本地，如图A-4所示。如果你想安装一个能在命令行中运行的工具，那就要用全局安装。这些模块会把可执行文件添加到你的执行路径（execution path）中。以Gulp为例：全局安装Gulp以后，你就可以在文件系统的任何地方，通过gulp命令来执行Gulp。全局模块是保存在模块空间（module space）里的，Node通过环境变量来引用它们。

如果你在你的软件中使用模块，那么你就应该用本地安装。然后你就可以在你自己的JavaScript文件中require这些模块，并使用它们的功能。在用Gulp的时候，你就会用到这个功能，因为所有用来转换文件的插件都是模块。

在编写Node软件的时候，有一条核心的原则：如果一个功能可以被做成模块，那就把它做成模块。这也就是为什么Node不像其他语言那样提供很多基本的功能。Node除了和操作系统相关的功能以外，不提供其他基本的功能。如果一个功能可以用JavaScript来写，那就应该用JavaScript来写它。

### A.2.2 全局安装的例子：LESS

为了理解Node.js背后的概念，我们用LESS写了一个例子。LESS是一个CSS预处理器。CSS预处理器让你可以用更高级的语法来写CSS，比如变量、mixin等。CSS

⊖ 作者原来附的地址已经不可访问了，但是想详细了解可以看这里：https://www.quora.com/I-keep-hearing-NPM-doesnt-stand-for-Node-Package-Manager-what-does-it-stand-for。——译者注

预处理器有很多，但是 LESS 是完全在 Node 用 JavaScript 运行的，而且它也是 NPM 仓库里被安装次数最多的工具之一。我们在第 2 章里已经安装了 LESS，并且用它来编译 LESS 文件了。LESS 是用 Node.js 写的，但是作为用户的你其实并不会注意到它所采用的技术。它的行为就和其他命令行工具一样。一个基本的 LESS 文件大概是下面的样子：

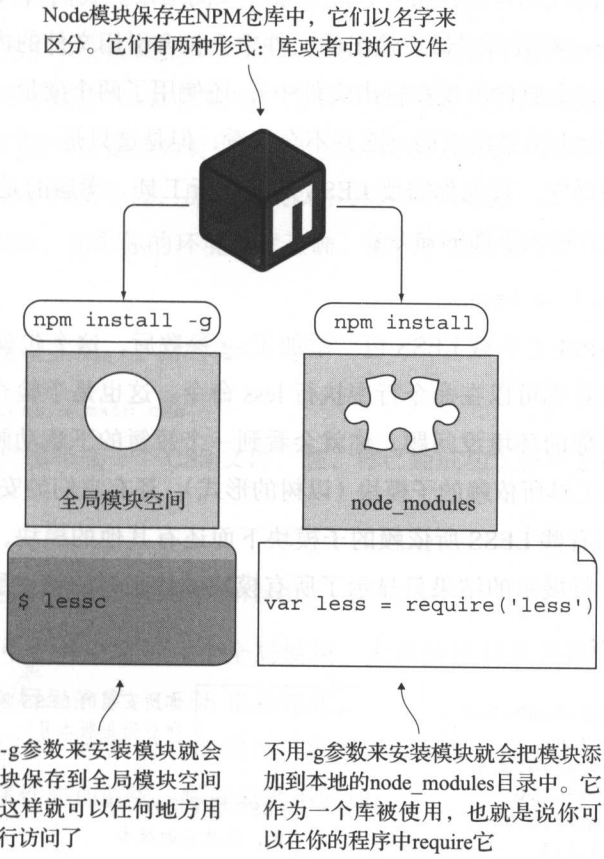



图 A-4　全局安装和本地安装。你的目标是创建一个可执行的应用。全局 Node 模块提供的是可执行程序，本地 Node 模块是在你自己的应用中作为库被使用的

```
@import (less) "normalize.css";   <-- 1 @import 规则在 CSS 中也有

@firstVariable: blue;             <-- 2 定义第一个变量
@secondVariable: red;

body {
  background-color: @secondVariable;   <-- 3 直接使用变量而不是值
  color: @firstVariable;
}
```

通过新（less）的记号，*你可以把任何 CSS 文件当成 LESS 文件来引用*❶。它的行为还是一样的，但是被引用的文件整个都会被包括进来。你还用 @ 来声明了变量，变量名完全由你决定❷。你可以给变量赋任何值。你可以在整个文件里复用这个变量❸，而且修改的时候只要修改定义它的地方就可以了。

这样写出来的代码可读性很好，而且很好预测结果。我们引入了外部的 CSS 文件 normalize.css（less 参数可以最后输出的文件中会包含引用文件的内容；如果没有这个参数，这行命令就会原样出现在输出文件中），还使用了两个变量。你可能已经注意到了这几行代码实际上渲染出来的页面并不会好看，但是这只是一个 demo。

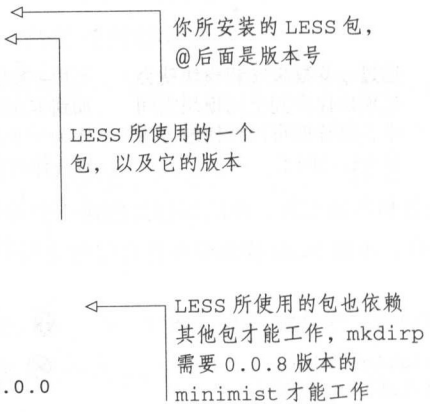
让我们来编译它。首先你需要 LESS 的命令行工具。幸运的是，NPM 上就有。启动你的终端，然后输入下面的命令：

```
$ npm install -g less
```

这个命令会从 NPM 上下载 LESS 包。添加了 -g 参数后，这个包就在全局范围内可用了。这也就意味着你可以在命令行中执行 less 命令。这也是个检查你的配置是否正确的好机会。如果你的环境没问题，你就会看到一个短暂的下载动画，然后会看到一系列被同时下载的工具所依赖的子模块（以树的形式），还有它们的安装位置。

你将会看到有些 LESS 所依赖的子模块下面还有其他的模块，那些是子模块所依赖的模块。命令行展示的结果只显示了所有模块的依赖树的第一层级，实际上的依赖树看起来像这样：

```
less@2.4.0
├?? graceful-fs@3.0.5
├?? mime@1.3.4
├?? image-size@0.3.5
├?? promise@6.1.0
?   ??? asap@1.0.0
├?? errno@0.1.1
?   ??? prr@0.0.0
├?? source-map@0.2.0 ()
?   ??? amdefine@0.1.0
├?? mkdirp@0.5.0 ()
?   ??? minimist@0.0.8
??? request@2.53.0
    ├?? caseless@0.9.0
    ├?? json-stringify-safe@5.0.0
    ├?? forever-agent@0.5.2
    ├?? aws-sign2@0.5.0
    ├?? stringstream@0.0.4
    ├?? oauth-sign@0.6.0
    ├?? tunnel-agent@0.4.0
    ├?? isstream@0.1.2
```



```
├?? node-uuid@1.4.3
├?? qs@2.3.3
├?? combined-stream@0.0.7
├?? mime-types@2.0.9
├?? tough-cookie@0.12.1
├?? http-signature@0.10.1
├?? form-data@0.2.0
├?? bl@0.9.4
??? hawk@2.3.1
```

所以，虽然 LESS 本身依赖很少，但是这些依赖还有别的依赖。这个依赖树可能比两层还要深，取决于其他 Node 模块的粒度有多细。甚至还有可能同一个包被安装了两次，但是版本却不同，就是因为有别的包分别依赖它们两个。但是 NPM 很聪明，如果已经下载过某个版本的某个包，那么下次就不会再次下载它了。

让我们回到 LESS。如果你的环境配置正确，输入下面的命令：

```
$ lessc --help
```

你就会看到 less 的帮助文档。下面的命令：

```
$ lessc main.less > main.css
```

将会把 main.less 文件编译成一个 CSS 文件。瞧，你已经成功了编译了你的第一个 LESS 文件了。

### A.2.3　本地安装的例子：Autoprefixer

在前面的例子中，你安装了一个全局模块。大多数项目不仅提供了命令行接口，同时也提供了用在项目里的库，LESS 也是如此。

让我们假设你用了一些新的 CSS 属性。你需要在属性前面加上浏览器前缀，但是你不知道现在哪些属性要加前缀，哪些属性不要加：

```
❶ 这是最标准的线性渐变的写法
header {
  background-image: linear-gradient(to bottom,black,white);
  background-image: -webkit-linear-gradient(top,black,white);
  background-image: -moz-linear-gradient(top,black,white);
}
```

❷ 这个属性作为实验属性已经很久了

❸ 对于 Firefox 也一样

第一行❶定义了渐变的方向和两个色标。有些基于 WebKit 的浏览器（大多数是移动端的）需要一个前缀（这里是 -webkit）才能使用这个特性。添加了第二行❷以后就

可以在这些浏览器中使用了。Autoprefixer会添加这一行。Firefox很快就实现了这个功能，虽然有些旧的Firefox也需要前缀才能用它，但它们几乎已经绝迹了。所以其实你并不需要添加第三行❸，因为大部分情况下，这一行是不会被解析的。Autoprefixer会删除这一行。

**浏览器前缀**

浏览器前缀是浏览器开发商用来实验新特性所规定的记号。浏览器开发商在JavaScript方法和CSS属性前加上特殊的缩写，来体现各个浏览器对它们的实现和语言标准之间的区别。比如说，-webkit-animation是添加了前缀的CSS动画属性，它会被WebKit的渲染引擎识别出来。

为了上面的效果，你需要运行一个叫作Autoprefixer的工具。Autoprefixer会从一个名叫CanIUse[⊖]的网站上获取浏览器前缀的信息，然后把需要的前缀添加到你的CSS属性前面。如果有的前缀已经不需要了的话，它还会删除它们，以保持输出结果的整洁。

Autoprefixer也是一个既有命令行版本，也有库版本的Node.js工具。当你全局安装它时，他是命令行版本；当你本地安装它时，他就是库版本。

如果你想用Autoprefixer来处理LESS代码，首先你要把LESS编译成CSS，然后把结果传递给Autoprefixer。这可以通过命令行来完成（或者LESS插件），但是你也可以写一段程序来实现这个任务。

要安装这两个库，首先进入到你的项目目录中，然后输入下面命令：

```
$ npm install less autoprefixer
```

注意两点：第一，不要加上-g参数，因为这次要本地安装。本地安装的模块无法在文件系统中的其他地方访问，只能在安装了这个模块的项目中使用。第二，install后面加了两个参数。也就是说，你可以用一个命令安装多个模块。当你安装完成后，你会注意到你的项目目录下多了一个名叫node_modules的目录，其中就是所有的模块。

下面的代码就是需要处理的LESS代码，把它保存到文件系统中的任意地方。

⊖ http://caniuse.com。

**代码清单 A-1　main.less**

```
@fore-color: blue;
@back-color: red;

body {
  background-color: @fore-color;

  header {
    background-image: linear-gradient(to bottom,@fore-color,@back-color);
  }
}
```

这段代码里用到了两个变量。初次以外，还用到了 linear-gradient。它在有的移动平台上是不支持的，要加上前缀。

要创建你的程序，首先在你安装模块的目录下面创建一个 index.js 文件，内容如下：

**代码清单 A-2　index.js**

```
'use strict';

var fs         = require('fs'),
    less       = require('less'),
    prefixer = require('autoprefixer');

var filename = (process.argv.length > 2) ? process.argv[2] : undefined;

filename && fs.readFile(filename, function(err, data) {

  if(err) throw err;

  less.render(data.toString(), function(err, output) {
    if(err) {
      console.log(err);
      throw err;
    }
    console.log(prefixer.process(output.css).css);
  });
});
```



你要执行命令将接受三个参数❶。命令是：node index.js main.less。main.less 就是第三个参数，在数组中的索引是 2。toString() 方法的作用是把你传递给 render 方法的文件内容转换成字符串❷。解析好 LESS 代码以后，把编译好的 CSS 代码传递给 Autoprefixer❸。最后输出的就是添加了前缀的 CSS 代码。

这个程序很短，即使你不熟悉 Node.js，也能猜出每一行代码的作用。毕竟它是 JavaScript。除了那些实际调用的方法以外，我还想让你关注下面几行代码。这几行代

码是用来require其他模块的。

```
var fs        = require('fs'),
    less      = require('less'),
    prefixer = require('autoprefixer');
```

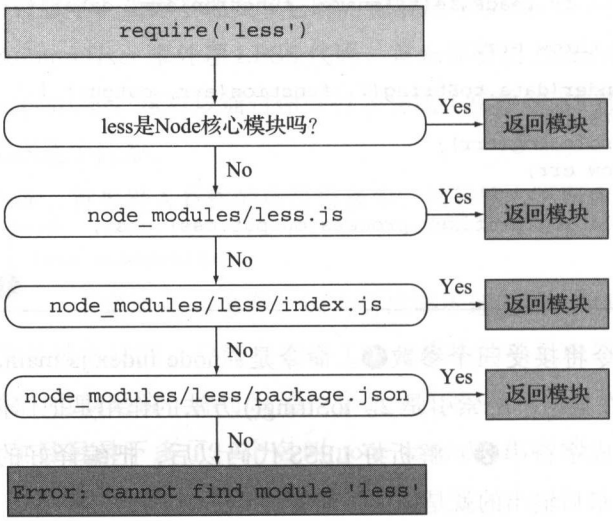
和A.1.3节中你require log10.js模块的时候相比，两者的代码中的参数有显著不同，这次的代码中的参数前有一个点和斜杠：

```
var log10 = require('./log10');
```

点和斜杠代表当前目录，整个字符串其实指向的是文件系统中的一个位置。如果你的参数中包括点或者斜杠，那么Node.js就会假设这个模块是在文件系统中的某个地方，并且会尝试加载它。如果参数中没有点或者斜杠，Node.js就会把它当成一个模块，然后依次按照图A-5中展示的顺序去寻找这个模块。

寻找的步骤如下：

- ❑ 这个模块是Node.js核心模块，比如fs或者http吗？
- ❑ 这是一个位于node_modules目录下的JavaScript文件吗？
- ❑ 这是一个node_modules目录下的一个含有index.js文件的目录吗？
- ❑ 这是一个node_modules目录下的一个含有包定义文件（package definition file）的目录吗？这个包定义文件是否指向模块的主文件？




图A-5 Node.js搜索包的流程图。当你require一个模块的时候，Node.js会依次判断这些条件，来找到正确的JavaScript文件

如果一条规则都不符合，Node 就会抛出错误。如果 Node.js 根据这些规则找到了对应的 JavaScript 文件，它就会加载这个文件并且把 API 暴露给调用者。最后一条规则有点特别，你可能还不知道什么是包定义文件，下一节我们就会介绍。

### A.2.4 用本地模块临时替换全局模块

无论你是全局安装还是本地安装一个模块，模块本身都是一样的。这意味着即使你本地安装一个模块，这个模块中也包含了可执行文件。虽然它们不在执行路径中，但是它们还是可以访问和执行的，你可以利用这个特性。如果你想要编译一个老项目里用旧版本 LESS 写的 LESS 文件的时候，你该怎么办呢？你或许不能用最新的 LESS，因为可能会有不兼容的改动，导致最后编译失败。

截止本书写作的时候，LESS 最新版本是 2.6.1。在 2012 年年底的时候有过一个 1.3.3 的版本。有些旧的 LESS 文件可能需要这个版本才能编译。幸运的是，你可以轻松地下载到这个版本：

```
$ npm install less@1.3.3
```

现在你可以在安装的目录中访问它的可执行文件：

```
 ./node_modules/less/bin/lessc –version
util.puts: Use console.log instead
lessc 1.3.3 (LESS Compiler) [JavaScript]
```

即使你已经全局安装了更新的 2.4.0 LESS，你仍然可以在本地运行更老的版本。你可以使用它的所有功能，同时也不会对全局安装的版本有任何影响。这样对于老项目而言就没有模块不兼容的风险了，你不需要升级整个环境，就可以安全地使用不同版本的工具。

## A.3 发布模块

在上一节中，你已经知道 Node 搜索模块的最后一个步骤是检查包定义文件，或者说是 package.json。尽管我们到目前为止还没有介绍过这个文件，但是它是我们接触了那么多模块以来最重要的文件之一，因为它包含了保证模块运行的所有信息。

### A.3.1 package.json 的内容

package.json 文件有两个作用：

- 定义所有 NPM 模块仓库所需要的包的元信息。
- 保存依赖、开发依赖的信息，NPM 根据这个信息知道要安装哪些模块。

让我们看一下 LESS 模块的 package.json 文件。我假设你是从 GitHub 上打开的[⊖]，因为那上面没有你发布模块时所产生的额外的元信息。这样看起来会更清楚：

```
{
  "name": "less",                                          ❶ 名称、版本和介绍
  "version": "2.4.0",
  "description": "Leaner CSS",
  "homepage": "http://lesscss.org",
  "author": { "name": "Alexis Sellier", "email": "self@cloudhead.net"},   ❷ 主页、作者和贡献者
  "contributors": [ "The Core Less Team" ],
  "bugs": { "url": "https://github.com/less/less.js/issues" },           ❸ 项目仓库和 bug
  "repository": {
    "type": "git",
    "url": "https://github.com/less/less.js.git"
  },
  "bin": { "lessc": "./bin/lessc" },                       ❹ 可执行文件
  "main": "index",                                         ❺ 主文件
  "engines": { "node": ">=0.10.0" },                       ❻ node 版本
  "scripts": { "test": "grunt test" },                     ❼ 脚本
  "dependencies": { ... }                                  ❽ 依赖
  "optionalDependencies": {                                ❾ 可选依赖
    "errno": "^0.1.1",
    "graceful-fs": "^3.0.5",
     ...
  },
  "devDependencies": {                                     ❿ 开发时依赖
    "diff": "^1.0",
    "grunt": "^0.4.5"
    ...
  },
  "keywords": [ … ]
}
```

首先定义的是名称和版本，这两个信息是用来在 NPM 仓库中区分模块的❶。当你自己开发模块的时候，你可以手动更新版本号。NPM 仓库会保存旧版本。之前已经讲过，你可以通过 npm install <packagename>@<versionnumber> 来安装特定版本的模块。❷这里是你可以自豪地炫耀的地方。❸这里是你的用户炫耀的地方。一般你可以在这里找到 GitHub 地址，你可以到 GitHhub 上反馈或者看源代码。你还可以在这里追踪 bug。如果这个模块是可以在命令行中运行的工具，那你可以在 bin 属性中定义指向可

⊖ https://github.com/less/less.js/blob/master/package.json。

执行文件的符号链接❹。你还可以定义多个可执行文件，来指向不同的文件。Node 会创建好这些可执行文件的。所有你想要全局安装的命令都要定义在这里。

当你要本地安装模块的时候，你可以定义模块的主文件❺。主文件通常是 index.js，但是你可以指向其他文件。和 bin 属性不同，这个属性不能是对象。engines 字段可以指定模块所支持的最低的 Node 版本❻。这里❼你可以定义一些通过 npm 命令在命令行中执行的快捷方式。在本例中，我们定义了 test，来代表 grunt test。只要通过 npm run test 就可以运行了。乍一看好像效率并不高，但是这个功能的目的不仅仅是充当快捷方式而已，更重要的目的是指定了软件使用你的模块的入口命令。在本例中，你可以通过这个命令来执行单元测试，但是你并不关心单元测试是怎么执行的。这对像 Travis ⊖这样的持续集成平台很有用。

❽依赖的作用简单，你的模块少了他们就没办法运行。NPM 会负责安装模块。如果它找不到某个模块，安装过程就会失败，你的模块也会被标记为不完整的。至于可选依赖，你的应用没有他们也可以运行，但是有了它们会得到额外的好处❾。NPM 会安装可选依赖，但是如果它们不能用了，NPM 还是会继续安装的。而开发时依赖则是在开发的时候才需要的依赖，比如构建系统、测试，或者其他类似的东西❿。这些依赖是不会被安装的，除非你指定 NPM 安装。

由于 package.json 文件保存了所有依赖的信息，所以在提交模块源代码或者发布到 NPM 仓库的时候，你不需要保存 node_modules 目录。

在 package.json 所在的目录中，运行一个简单的 npm install 命令，就可以重新安装所有依赖的指定版本。有时候你会升级某个依赖（使用 npm update <packagename>），结果发现这个新版本和其他模块不兼容。这个时候 package.json 很有用了。你只要删除 node_modules 目录，并且用上面这个简单的命令就能重新安装所有依赖了。

还要注意一点，依赖的版本号都注明在了包的名称后面。你可能注意到了版本号前的符号 ^。

```
"devDependencies": {
  "diff": "^1.0",
  "grunt": "^0.4.5"
  ...
},
```

这个符号当然是有意义的。

⊖ http：//travis-ci.orq。

## A.3.2 NPM 处理版本的方法

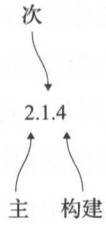


图 A-6 版本系统

当你第一次安装一个模块，并把它保存到 package.json 中时，NPM 会在这个模块的版本号前加上一个 ^。这个符号会告诉 NPM 这个版本就是一开始安装的版本，但是如果有更高的版本的话，也可以使用更高的版本，只要符合兼容性就可以。为了说明怎么样才算符合兼容性，我们要简单介绍一下由 GitHub 的人们创造的版本系统（versioning system）SemVer[⊖]，如图 A-6 所示。

版本号通常由三部分构成：主版本号（第一个数字），次版本号（中间的数字）和构建版本号（最后一个数字）。由这三部分构成的版本号很常见，但是 SemVer 在此之上还有一套规则。

这套规则很复杂，但是总体而言有三个原则，定义如下：

- 每次你向软件引入重大的、不兼容的更新时，都要更新主版本号。这表示新版本的 API 和旧版本已经不兼容了。
- 每次你向软件添加新功能，但是新版本同时向后兼容时，都要更新次版本号。你的 API 没有变，但是内部工作机制改变了。你还可以在这里引入即将被弃用的 API，然后在下个主版本号更新时彻底移除那个 API。
- 每次你修复 bug 时，都要更新构建版本号，这些改动往往是不重要的。

总结一下就是：每次构建都必须要更新版本号。如果添加新功能，就要更新次版本号。如果你改动了主版本号，那你接下来几周可能都不会接别的任务了。

NPM 也是按这套规则来的。这也就是为什么 package.json 的版本号前会有符号来表示决定到底使用哪个版本。

添加一个波浪线符号（~）表示安装和版本号相近的版本。比如对于 ~1.2.3，NPM 就会安装从 1.2.3 到 1.3.0（不包括 1.3.0）范围之内的版本。指数符号（^）表示安装和版本号相兼容的版本。这会把次版本号也考虑进去。所以对于 ^1.2.3，NPM 就会安装从 1.2.3 到 2.0.0（不包括 2.0.0）范围之内的版本。

如果你已经下载了一个包，那么当你再次执行 npm install 的时候，即使这个包有了更新，NPM 还是会使用已经安装过的那个版本。

图 A-7 展示了类似的过程。如果你后来移除了版本号，而且安装了更高的版本，

⊖ Semantic Versioning 的缩写：http://semver.org。

NPM 还是会把原来那个版本保存在项目中：

1.2.0 <= ~1.2.4 < 1.3.0

1.0.0 <= ^1.2.4 < 2.0.0

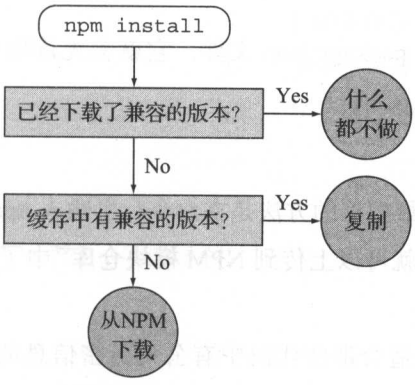



图 A-7　NPM 是如何下载包的。当你调用 npm install 的时候，它会检查是否已经下载了兼容的版本。如果没有，就会检查缓存中有没有。如果还是没有，它才会去检查 NPM 仓库

SemVer 在 Node.js 中很管用，但是它有一个不足之处：如果你的主版本号是 0，那么代表你的软件还是测试版，你的 API 也会被认为是不稳定的。这就告诉开发者在使用这个包的时候，它的接口可能随时会变，而且等到这个模块发布 1.0 版本时，API 可能完全不同了。

### A.3.3　创建模块并发布

如果你想把软件打包成一个模块，你要做两件事：

- 提供 package.json。
- 发布到网上。

所以首先你要创建一个 package.json 文件来保存依赖和你的个人信息，你可以采用下面的方法：

- 创建个新文件，然后自己手动输入内容。
- 根据已有的文件修改。
- 使用 NPM 提供的工具。

如果你输入

```
$ npm init
```

你就会看到一个简短的问卷，这个问卷设计得很智能。如果你已经建立了一个 Git 仓

库，那它就会自动添加有关的信息。如果你在项目根目录里已经创建了主文件，它就会自动把文件名添加进去（如果没有，它就会默认在根目录创建一个index.js）。它还会保存所有已经下载和安装的Node模块作为依赖。一切完成后，你也可以单独修改package.json文件。

如果你已经有了一个package.json文件，它就会先读取它，然后再询问那些不在package.json中的信息。

#### 发布

现在是时候发布了，最简单的方法是在命令行里输入npm publish命令。如果模块名称没有重复，你的模块就可以上传到NPM模块仓库[⊖]中了，之后人们就可以搜索和安装了。

还有一种方法，比较适合那些代码中有公司保密信息的模块（但实际上你不应该在代码里添加这些信息）：你可以用Git仓库来保存模块。安装的时候在npminstall后面加上Git URL作为参数就可以了。这样一来你即使不把模块上传到NPM仓库中，也可以把它当成模块来用，只不过没办法搜索它就是了。

package.json文件是Node.js模块的关键，它保存了版本号、依赖以及主文件的信息。熟悉了前几节介绍的知识后，你在开发Node.js软件时基本上就不会遇到障碍了。

## A.4 总结

在附录中，你学习了本书所涉及的Node.js的基本工作原理：

- Node.js的生态系统包括运行时、核心、模块和你自己写的程序。
- Node.js模块是通过NPM来发布和分发的。
- 模块可以全局安装，也可以本地安装。如果你全局安装一个模块，它就会被作为工具使用；如果你本地安装一个模块，它就会被作为库来使用。
- 一个模块被本地安装，当成库来使用时，这个模块一定要有一个index.js文件，或者在package.json中指定别的主文件。这样才能通过require来引用这个模块。
- Node模块必须要提供package.json文件。package.json文件包含了模块所需的所有必要的元信息，以及模块的依赖。

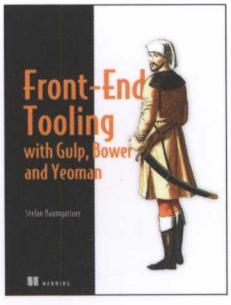
---

⊖ http://www.npmjs.org。

## 作者简介

**斯特凡·鲍姆加特纳（Stefan Baumgartner）**

在奥地利林茨生活和工作。他是一位非常热情的Web开发者和技术布道师，经常在当地组织技术分享和会议。他的研究领域涵盖Web性能、自动化、架构和渐进式增强。




原书封面

# Front-End Tooling

with Gulp, Bower and Yeoman

# Web前端自动化构建 Gulp、Bower和Yeoman开发指南

唯一一本完整涵盖前端工具的书。

——Palak Mathur, Capital One

提供了快速从新手到专家的所有信息。

——Jason Gretz, Auto-Owners Insurance

本书包括了你所需要的所有前端开发工具。

——Unnikrishnan Kumar, Thomson Reuters

本书是目前最易懂、最实战的构建前端应用的指南。

——Nick A. Watts, American Chemical Society

在大型Web项目的开发中，工作流决定了生产力。Gulp、Bower和Yeoman可以把从开发到构建再到部署的整个工作流自动化，这些强大的工具结合所形成的构建系统可以大大缩短Web应用的发布周期。

本书教你如何搭建自动的开发工作流。首先你会形成一个对开发过程的整体认识。然后，本书会通过例子深入指导你如何用Gulp、Bower和Yeoman构建一个产品发布流程。读完本书后，你将深入理解Web开发流程，并掌握用最优秀的工具开发强大、自定义的工作流的能力。

MANNING

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：程序设计 / Web开发

ISBN 978-7-111-57883-3

9 787111 578833

定价：59.00元